Machete

Thwarting Code-Reuse Attacks Through Temporal Permission Tightening in User Space

Yigit Colakoglu

Delft University of Technology

Machete

Thwarting Code-Reuse Attacks Through Temporal Permission Tightening in User Space

by

Yigit Colakoglu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday April 17, 2026 at 15:00

Student number:	5495121	
Project duration:	October 2025 – April 2026	
Daily supervisor:	Alexios Voulimeneas	
Thesis committee:	Alexios Voulimeneas	TU Delft
	Georgios Smaragdakis	TU Delft
	Soham Sundar Chakraborty	TU Delft

Cover: Power Surge by Katerina Belikova.

TU Delft

Nomenclature

Abbreviation	Definition
W $\oplus$ X	Exclusively Writable or Executable Memory
XoM	Execute-Only Memory
PKU	Protection Keys Userspace
MPK	Memory Protection Keys
SUD	Syscall User Dispatch
JIT	Just in Time (Compilation)
FSM	Finite State Machine
PTA	Prefix Tree Acceptor
EDSM	Evidence-Driven State Merging

Preface

This document contains the Master Thesis to fulfil the graduation requirements for the program in Computer Science at Delft University of Technology. In the process of writing it, I had the opportunity to explore the outer fringes of systems security and got a glimpse into what being a researcher in this field would be like.

I tried to incorporate the experience I gained in the industry and in CTFs with the theory I learned during my studies. The idea was to create a framework with strong theoretical grounding that also considered what makes a security tool useful in practice, while remaining transparent about both its protections and its limitations.

I was surprised, however, by how commonplace it has become to develop security systems that rely on superficial attacker models. Security is one of the few fields where every sentence ends with an exception footnote[1], and I believe it is especially important in academia to not omit those footnotes. As such, I tried to give a thorough discussion of both the upsides and downsides of my solution.

Writing this thesis was an intense and humbling experience, and I would not have been able to finish it if it had not been for the people who kept me grounded when the research felt intractable. As such, I would like to start by thanking Prof. Alexios Voulimeneas and Vissarion Moutafis for the invaluable feedback they provided along the way.

This thesis was likely the longest and most demanding undertaking I have ever set out to complete, during which I encountered many achievements and setbacks. I am thankful to everyone who has been by my side through those moments. I would also like to especially thank my dear friends Orhan, Kerem, Kaan and Konstantin for putting up with my endless trains of thought and sharing their invaluable input.

Last but not least, I would like to thank my family for their unconditional love and support throughout this journey. I would especially like to thank my mother, Sibel, and my father, Tugrul, who technically (by the transitive property of having made me) are co-authors to this work. My gratitude towards them is more than I can put into words.

Yigit Colakoglu
Delft, April 2026

[1] Of course, there are exceptions to this

Abstract

Modern software ships with substantial unused code. Dynamic linking loads entire shared libraries when only a fraction is required, features accumulate over release cycles, and one-size-fits-all distribution models ship complete binaries regardless of deployment context. This bloat directly expands the attack surface available to adversaries: unused but mapped code provides gadgets for return-oriented programming (ROP) and jump-oriented programming (JOP) attacks. Existing defenses are partial. Address space layout randomisation is defeated by memory-disclosure vulnerabilities. Execute-only memory prevents reading code pages but does not reduce their volume. The W⊕X policy prevents code injection but not code reuse.

Software debloating removes unused code, but existing tools face tradeoffs between source access, soundness, precision, and deployment requirements. Binary-level tools such as Razor operate only on the application binary and leave shared libraries fully mapped. Static-analysis tools such as Decker are conservative in their approximation and likewise skip library code. Kernel-level mechanisms can achieve strong isolation but require kernel modifications that limit deployment. No existing system combines temporal restriction, where different code is accessible at different stages of execution, with execute-only memory enforcement over the full dependency chain in user space.

We present Machete, a software debloating framework that derives temporal memory-access policies from execution traces and enforces them entirely in user space, without kernel modifications or source code access. Machete operates in three stages. A segfault-based profiler captures page-granularity access patterns for both single- and multi-threaded programs. A modified Blue-Fringe / EDSM learner infers a phase-structured finite-state machine from these traces, with tunable scoring coefficients that control the security/performance tradeoff. An enforcement runtime then runs each phase in a separate operating-system process with its own page-table permissions and execute-only memory over shared physical memory, debloating the full dependency chain including shared libraries.

We evaluate Machete against Razor and Decker on 11 shared targets and 2 auxiliary multi-threaded targets. Machete reduces executable pages by 86 to 95% from the original binary; even the worst-case phase exposes 2 to 5 times fewer pages than both Razor and Decker. ROP gadgets drop by 33 to 78% per phase, and all enforced variants have zero readable-executable application pages, preventing runtime gadget discovery. For CPU-bound workloads, enforcement overhead is below 4%. For server workloads, overhead ranges from 2.7% (memcached) to 66.6% (lighttpd) depending on phase-transition frequency, and a parameter sensitivity analysis confirms that the operator can navigate this tradeoff through stable regions in the scoring-coefficient space. Machete is, to our knowledge, the first system to combine temporal debloating of executable pages with execute-only memory enforcement in user space.

Contents

1 Introduction

Modern software systems contain substantial amounts of unused code. This bloat arises from prevalent software engineering practices: dynamic linking loads entire libraries when only small parts are used, features accumulate over time through scope creep, and one-size-fits-all distribution models ship complete functionality regardless of individual user needs. Empirical studies have quantified this phenomenon: Quach et al. found that only 5% of the GNU C Library is used across over 2,200 programs in a standard Ubuntu Desktop installation, with even the heaviest user (the VLC media player) requiring merely 18% [1]. This bloat is not just a performance concern but a significant security risk. Libraries are often loaded in their entirety when only a subset is required, increasing the deployed dependency footprint and thereby exposing applications to vulnerabilities anywhere in that footprint, as illustrated by high-severity flaws in widely deployed dependencies such as OpenSSL (Heartbleed) [2], log4j (Log4Shell) [3], and ImageMagick [4].

Unused code also directly expands the attack surface available to adversaries. In general, each unnecessary function provides an additional potential target for exploitation and can increase the pool of gadgets (small instruction sequences that attackers chain together in return-oriented programming (ROP) and jump-oriented programming (JOP) attacks). Defenses against such code-reuse attacks exist but have fundamental limitations. Address space layout randomization (ASLR) randomizes code locations to prevent attacks relying on hardcoded addresses, yet memory disclosure vulnerabilities allow attackers to discover these locations at runtime. Just-in-time ROP (JIT-ROP) exploits such disclosures to scan code pages dynamically, build gadget chains on the fly, and execute exploits without prior knowledge of binary layout [5]. Execute-only memory (XoM) prevents this by denying read access to code pages, but it does not reduce the attack surface: all code remains mapped and executable.

Software debloating techniques aim to address this problem by removing unnecessary code. However, existing approaches face significant trade-offs. For this thesis, the most important comparison axes are source availability, soundness, policy precision, and deployment requirements. Source-level and compiler-based techniques require access to source code, excluding commercial off-the-shelf (COTS) binaries. Binary-level approaches such as RAZOR rely on execution traces but are unsound; code paths never exercised during profiling are removed, causing crashes if encountered at runtime [6]. Static analysis approaches such as Decker partition functions into "decks" enabled on demand, but conservative call-graph approximations limit precision and require compiler-level analysis [7]. Kernel-level mechanisms including TLASR and SMV provide strong isolation but require kernel extensions or manual programmer specification [8,9]. Taken together, these limitations leave a gap for a system that can operate on unmodified binaries, adapt code permissions over time, resist code-disclosure attacks, and do so entirely in user space.

This thesis presents Machete, a software debloating framework that combines debloating with execute-only memory to both reduce the volume of executable code and hinder direct code disclosure. Machete derives temporal memory-access policies from execution traces and enforces them entirely in user space, without kernel modifications or source code access. A temporal policy recognizes that programs require different code at different points in their lifecycle, for instance during initialization versus steady-state operation, enabling finer-grained attack surface reduction

than static approaches. Concretely, this work is guided by one main research question and three supporting subquestions:

- **Main RQ:** To what extent can temporal memory-access policies reduce the executable attack surface of unmodified binaries without kernel modifications?
  - **RQ1:** To what extent can execution traces be transformed into compact policies that generalize across different executions of the same program for temporal software debloating?
  - **RQ2:** To what extent does combining debloating with execute-only memory reduce exposed executable pages and gadget availability for code-reuse attacks?
  - **RQ3:** What runtime overhead does user-space enforcement of temporal memory policies incur compared to unprotected execution?

Machete operates through a three-stage pipeline. First, it profiles a target binary and records which file-backed code pages are accessed over time. Second, it learns a compact phase model from these traces, where each phase captures a different part of the program's execution and transitions are triggered by new page accesses or system calls. Third, it enforces the resulting policy by running each phase in a separate user-space process with its own page permissions over shared memory and switching between phases as execution progresses. Because all of this happens in user space, Machete requires no kernel modifications and can be combined with complementary analyses such as static analysis or compiler annotations.

The remainder of this thesis is organized as follows. Chapter 2 provides background on software debloating, execute-only memory, hardware isolation primitives, and automata learning. Chapter 3 describes the design and implementation of Machete's profiling, policy derivation, and runtime enforcement stages. Chapter 4 presents the evaluation methodology and results. Chapter 5 concludes with a discussion of limitations and future work.

Background and Related Work 2

This chapter covers the research areas that Machete builds on: software debloating, execute-only memory, in-process isolation with hardware primitives, and behavioural modelling through automata learning.

2.1 Software Debloating

Modern software ships far more code than any single deployment actually uses. Quach et al. [1] show that, on average, only 5% of the GNU C Library is used across over 2,200 programs in a standard Ubuntu Desktop installation; even the heaviest user, the VLC media player, only needed 18%. This *code bloat* [10] comes from common practices like code reuse, feature accretion, and one-size-fits-all distribution. Bloat is often discussed as a performance problem (larger binaries, slower startup, more memory usage), but its security impact is just as important. Every function that is shipped but never called increases the attack surface: unused code may contain vulnerabilities and gives attackers a larger pool of instruction sequences, or *gadgets*, to chain into return-oriented programming (ROP) [11] and jump-oriented programming (JOP) [12] attacks.

Recent systematisation-of-knowledge papers give a broad overview of the field. Alhanahnah et al. [13] develop a taxonomy that classifies debloating tools by their input and output artefacts, strategies, and evaluation criteria. Ali et al. [14] add an empirical comparison of debloating paradigms and their compositions, while Brown et al. [15] compare the attack-surface reductions that individual tools claim against the reductions measured under unified benchmarks. Together, these surveys show that existing debloating techniques cover a wide range of trade-offs between how aggressively they remove code and what soundness guarantees they can offer.

2.1.1 Source-Level and Compilation-Based Debloating

A natural place to remove unused code is the compilation pipeline, where the compiler has access to full semantic information such as types, control-flow graphs, call graphs, and data-flow relations. OCCAM-v2 [16] performs whole-program specialisation on LLVM intermediate representation by combining pointer analysis, value analysis, and dynamic profiling to remove provably unreachable functions. BLADE [17] builds program-dependence graphs at the source level to find and remove code that does not contribute to a given set of target features. Koo et al. [18] observe that many applications expose configuration parameters that implicitly define which code paths can run; their system uses known settings to remove code that is unreachable under a given configuration. More recent work explores data-driven strategies: Porter et al. [19] combine static analysis with machine-learning classifiers to predict the dynamic callee set at each indirect call site, restricting which functions can be invoked at runtime, and Chisel [20] treats debloating as a reinforcement-learning problem where an agent iteratively removes source-level statements while a test suite checks correctness.

Source-level and compilation-based approaches benefit from precise analysis but share one key limitation: they need the application's source code or intermediate representation. This rules out commercial off-the-shelf (COTS) binaries, third-party libraries that are only available in compiled form, and legacy software whose source code is lost. In addition, static analysis must conservatively keep code that *may* be needed under some input, which limits how much code can be removed.

2.1.2 Binary-Level and Post-Deployment Debloating

Binary-level debloating works on compiled executables and does not need source access, making it applicable to the COTS binaries that dominate real-world deployments. The main challenge is the lack of semantic information: without type annotations, symbol tables, or high-level control-flow constructs, the analysis must either be conservative or risk breaking the program.

RAZOR [6] is a representative trace-based approach. It profiles the application under user-supplied test inputs, identifies the executed code, and uses control-flow heuristics to keep additional code that may be reachable from the observed paths. Despite these heuristics, RAZOR is not sound: code paths that were never exercised during profiling are removed, and the debloated binary will crash if those paths are encountered at runtime. The system therefore depends on having a comprehensive test suite, which is hard to guarantee in practice.

Decker [7] is the closest prior work to Machete. It uses static analysis to partition a program's functions into *decks*, i.e., groups of functions that should be accessible together during an execution phase. A runtime component enables and disables decks on demand as the program moves between phases. On the SPEC CPU 2017 benchmarks, Decker achieves a 73% reduction in reachable gadgets with about 5% overhead, and its transformation is sound: all functions needed by a phase are enabled before use. However, its reliance on static analysis for phase identification limits precision. Conservative call-graph approximations can group unrelated functions into the same deck, and complex or indirect control flow, common in event-driven applications, is hard to resolve statically. Moreover, its `mprotect` based approach to control execution permission is not suitable for multi-threaded applications. Decker also requires compiler integration and operates at function granularity, whereas Machete learns phases dynamically from execution traces, works on unmodified binaries, and enforces permissions at page granularity.

Several other binary-level systems exist. LeanBin [21] lifts binaries to an intermediate representation, debloats at that level, and recompiles, but the lifting step is fragile and can fail on complex binaries. DamGate [22] uses dynamic adaptive gating to selectively enable features at runtime. Mansouri et al. [23] target functions associated with specific Common Vulnerabilities and Exposures (CVEs) and disable them in deployed binaries, while Huang et al. [24] debloat closed-source Windows applications. Across all these approaches, a tension remains between soundness and aggressiveness: systems that guarantee correctness tend to keep too much unused code, while more aggressive systems risk crashes. To the best of our knowledge, no prior binary-level debloater learns only behavioural models from execution traces to guide its reduction.

2.1.3 Library Debloating

Shared libraries are a major source of code bloat because linking against a library makes all of its code accessible, even if the application only uses a small part. Several approaches target this problem. Nibbler [25] does function-level debloating of binary shared libraries by removing functions that are not reachable from the application's call sites. Quach et al. [1] propose Piece-Wise, a compiler-assisted technique that loads library components independently so that unused parts are never mapped. D-Linker [26] relinks shared libraries from their object files, leaving out objects that contribute no symbols the application uses. Ziegler et al. [27] generate per-application library variants containing only the needed functions, while LibFilter [28] achieves a similar result through binary recompilation of `.so` files. Zhang et al. [29] extend library debloating to resource-

constrained embedded systems based on the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture.

Library debloating reduces the amount of mapped code but does not address bloat in the application binary itself. It also does not account for how an application's code needs change over time: a library function may be needed during initialisation but not during steady-state operation, and static library debloating cannot capture this distinction.

2.1.4 Kernel, Container, and Domain-Specific Debloating

Debloating has also been applied beyond user-space applications. Hacksaw [30] debloats the kernel by inventorying the devices on a machine and removing code for absent hardware. Zhang et al. debloat at the container level using filesystem-layer analysis [31], and at the framework level, targeting unused code in machine-learning toolkits such as PyTorch and TensorFlow [32]. Although these efforts are not directly relevant to binary debloating, because they show that debloating is a broader systems problem rather than one confined to application binaries. However, they do not address the setting studied in this thesis: user-space binary debloating with runtime-enforced temporal memory permissions.

2.1.5 Thread-Level and Context-Aware Debloating

More recent work recognises that different threads in the same process play different roles and need access to different parts of the code. Rommel et al. [8] introduce Thread-Level Attack-Surface Reduction (TLASR), which uses the `mmview` Linux kernel extension to give each thread its own view of the text segment. By removing code that a thread never needs, TLASR reduces the executable code visible per thread by 84% to 98% in benchmarks including MariaDB, Memcached, OpenSSH, and Bash. The gadget reduction makes an automated ROP-chain generator ineffective in all tested cases. The key finding is that worker threads handling client connections only need 3–5% of the total functions loaded into the process, which directly motivates the phase-based memory views that Machete enforces.

However, TLASR requires both a custom kernel extension and developer-supplied source annotations to define and mark thread boundaries. Machete learns phase transitions dynamically from execution traces, works on unmodified binaries, and needs no kernel changes. Both systems achieve per-thread or per-phase code reduction, but through different mechanisms: TLASR uses developer-annotated thread-role identification with kernel-level enforcement, while Machete uses learned behavioural models with user-space process isolation.

2.2 Execute-Only Memory

Even after debloating removes unused code, the code that remains can be read by an attacker with a memory-disclosure primitive. Just-in-time ROP (JIT-ROP) attacks exploit this: given an arbitrary read, the attacker scans code pages at runtime to find gadgets, builds an ROP chain on the fly, and diverts control flow, all without knowing the binary layout beforehand. Execute-only memory (XoM) prevents this by denying read access to code pages while keeping them executable, so the attacker cannot scan for gadgets.

2.2.1 Foundations and Implementations

Backes et al. [33] first showed that XoM can defeat JIT-ROP on x86. Because x86 page tables do not have a native execute-only permission bit (the execute permission implies read), their system approximates XoM in software by marking code pages non-present and intercepting read attempts in the OS page-fault handler. NORAX [34] brings XoM to COTS binaries on AArch64, where the hardware does support execute-only permissions, by separating embedded data from code sections. HideM [35] achieves XoM in user space through split page-table manipulations. Lixom [36] utilizes XoM to protect cryptographic keys by encoding them as immediate values in execute-only instructions, using Extended Page Tables (EPT) for strong isolation and Memory Protection Keys (MPK) for a lightweight variant. Luo et al. [37] tackle the challenge of adding XoM to stripped binaries that lack relocation metadata.

The main limitation of all XoM mechanisms is that they protect code confidentiality but do not shrink the attack surface: all code pages stay mapped and executable. An attacker who cannot read code can still execute it. To combine XoM with debloating so that only the pages needed in the current phase are mapped and executable, dynamic permission management is required, which neither XoM nor debloating alone provides.

2.2.2 Attacks on and Limitations of XoM

XoM does not guarantee absolute protection. Schink et al. [38] evaluate how well XoM holds up in practice and show that implementation artefacts can leak information about execute-only pages. Hornetz et al. [39] show that port-contention side channels can identify inaccessible code regions, and Bhat et al. [40] propose ProbeGuard, which detects and mitigates probing attacks on XoM pages. These results show that XoM alone is not enough and must be combined with attack-surface reduction and runtime enforcement.

2.3 In-Process Isolation and Hardware Primitives

Enforcing per-phase or per-thread memory permissions at runtime requires a mechanism that can switch permissions frequently and with low overhead. MPK, least-privilege memory views, and system-call interposition are the three building blocks that Machete combines for this purpose.

2.3.1 Memory Protection Keys (MPK)

MPK, available on recent Intel and AMD processors, assigns a 4-bit protection key to each virtual page. The Protection Key Rights for User pages (PKRU) register is thread-local and writable from user space via the `WRPKRU` instruction in roughly 20–100 cycles, and it controls read and write permissions per key domain. This makes MPK the fastest way to toggle memory permissions without entering the kernel.

ERIM [41] is the best-known MPK-based isolation system. It combines MPK with binary inspection to make sure untrusted code cannot run `WRPKRU` outside designated call gates, and achieves less than 1% overhead at 100,000 domain switches per second. Voulimeneas et al. [42] study the attack surface that MPK itself creates: if untrusted code can run a `WRPKRU` instruction, it can turn off all protection-key restrictions. Their work shows that binary inspection or control-flow integrity is needed to prevent such bypasses. Jenny [43] tackles a related problem: even when MPK correctly isolates memory domains, an attacker in an isolated domain can still use system calls to escape. Jenny prevents this by enforcing per-domain syscall policies.

Other MPK-based systems explore different aspects of in-process isolation. ThreadLock [44] provides per-thread MPK domains, TME-Box [45] looks at Intel's Total Memory Encryption with Multi-Key as an alternative to MPK, Gülmez et al. [46] use MPK to isolate Rust `unsafe` code, and Wang et al. [47] use MPK to protect in-process monitoring infrastructure.

A key limitation of MPK for Machete is that it only controls read and write permissions; it *cannot* control the execute permission. To restrict which pages are executable, which is at the core of Machete's enforcement, we need page-table-level mechanisms such as `mprotect` or, as Machete uses, per-process page tables with separate execute permissions.

2.3.2 Least-Privilege Memory Views

The principle of least privilege, applied to memory, means that each part of a program should only be able to access the memory it currently needs. Secure Memory Views (SMV) [9] is a direct ancestor of Machete's design. SMV introduces *thread containers* that split a multithreaded application's address space into memory domains, each with one of four privilege levels: read, write, execute, or allocate. Enforcement happens at the kernel level via per-thread page tables, with negligible overhead (around 2% on PARSEC, less than 1% on web servers). However, SMV requires the programmer to manually define domain boundaries and assign threads to domains, which limits its practical use.

Wedge [48] is an early system for privilege-separated compartments that motivated much of this line of work. TLASR [8] provides per-thread text-segment views using kernel-level `mmview` extensions, and μSwitch [49] achieves low-overhead context isolation by deferring domain switches to the next system call boundary, avoiding per-switch kernel involvement.

The shared limitation of these approaches is where the isolation policy comes from: it is either specified manually (SMV, Wedge), or derived from static analysis (TLASR). Moreover, some of them (TLASR, SMV) require kernel modifications. None of them learns a phase-based policy from actual execution behaviour. Machete fills this gap by deriving its policy automatically from execution traces using automata learning and enforcing it entirely in user space without kernel changes.

2.3.3 System Call Interposition

Observing and controlling a program's interactions with the operating system is needed for both profiling and enforcement. Traditional interposition mechanisms have significant overhead or expressiveness limitations: `ptrace` requires two context switches per intercepted call (one to enter the tracer, one to resume the tracee), making it impractical for production deployments, and `seccomp-BPF` can only filter calls by number, not modify their arguments or observe their results. GHUMVEE [50] applies `ptrace`-based interception to synchronise multi-variant execution for intrusion detection, demonstrating the security value of syscall monitoring while also illustrating the overhead penalty that `ptrace` imposes.

Binary rewriting offers a lower-overhead path. Zpoline [51] replaces every `syscall` instruction with a `callq *%rax` trampoline that redirects all system calls through a user-space hook, eliminating per-call kernel involvement and achieving overhead close to that of an ordinary function call. Lazypoline [52] extends this with lazy rewriting: rather than scanning and patching all syscall sites up front, it defers patching to first use, locating and rewriting each site on demand to reduce initialisation cost while preserving the same low per-call overhead.

Machete's interposition layer is built on top of K23 [53], which comes from this lineage by combining the Linux Syscall User Dispatch (SUD) mechanism with zpoline-style instruction

rewriting. During profiling, K23 intercepts memory-management calls such as `mmap` and `mprotect` that would otherwise escape the segmentation-fault-based tracer; during enforcement, it detects phase transitions triggered by system calls. Blair et al. [54] take a different direction, automatically synthesising per-microservice syscall policies from effect graphs, and Rosti et al. [55] use syscall-level replication across variants to maintain service availability. These systems demonstrate the breadth of syscall interposition applications, but none uses interposition to derive or enforce per-phase memory-access policies.

2.4 Behavioural Modelling and Automata Learning

Debloating requires a policy that says which code is needed and when. Static analysis can approximate such a policy, but its precision is limited because general program properties are undecidable. An alternative is to learn the policy from observed executions, trading formal soundness for practical precision.

2.4.1 Learning from Execution Traces

Execution traces, i.e. ordered sequences of events recorded during program runs, are a rich source of information about how an application behaves. Forrest et al. [56] showed that short-range correlations in system-call sequences define a stable notion of *normal* behaviour for Unix processes, and that deviations from this baseline can detect intrusions. The idea that regular patterns in low-level event streams characterise an application's behavioural modes is the starting point for Machete's phase model.

Later work has refined trace-based modelling. Multi-variant execution (MVX) systems compare system-call streams across multiple diversified program variants to detect divergences caused by exploitation; GHUMVEE [50] and A8 MVX [55] are representative systems in this family. Walkinshaw et al. [57] infer extended finite-state machines (EFSMs) from software executions, adding data guards and state variables to capture richer behaviour than plain finite automata. Their formulation of state transitions triggered by observable events is directly relevant to the phase-transition model that Machete learns.

A gap in this body of work is that trace analysis has been used for software testing, anomaly detection, and multi-variant execution, but not for deriving runtime security policies that restrict memory access. Machete closes this gap by using execution traces not just to characterise behaviour but to build a deterministic finite-state machine whose states directly encode per-phase memory permissions.

2.4.2 Automata Learning Theory

The formal foundations for learning finite-state machines from observations come from the field of grammatical inference [58]. De la Higuera [59] gives a thorough overview of the theory and algorithms. The central idea is *state merging*: given a prefix-tree acceptor (PTA) built from positive examples, states are iteratively merged to produce a compact automaton that generalises beyond the training data. Oncina and García [60] introduced Regular Positive and Negative Inference (RPNI), the first polynomial-time state-merging algorithm, which merges states in an order fixed by a breadth-first traversal of the PTA. The Blue-Fringe variant, later refined into Evidence Driven State Merging (EDSM), improves on RPNI by using a RED/BLUE frontier to control which merges are tried first and by scoring merges based on the evidence that supports them [61].

Machete's policy-derivation stage builds directly on this framework. We build a PTA from the execution traces collected during profiling and run a modified Blue-Fringe / EDSM learner to produce a compact deterministic FSM. The main difference from standard automata learning is that our states are not abstract language classes but concrete execution phases with associated memory-permission sets, and our transitions are triggered by newly observed page accesses or system calls rather than input symbols. To the best of our knowledge, automata learning has been used for software testing and program comprehension but not for deriving runtime memory-isolation policies.

2.5 Control Flow Integrity

Control-flow integrity (CFI) [62] restricts the targets of indirect branches to legitimate destinations, preventing an attacker from jumping to arbitrary gadgets. Zhang et al. [63] show CFI enforcement for COTS binaries, Payer et al. [64] refine CFI policies to reduce the set of allowed targets per indirect branch, and ShadowGuard [65] adds return-address protection through a cryptographic shadow stack.

CFI and debloating are complementary. CFI constrains *how* remaining code can be reached but does not reduce *which* code is present; debloating reduces the available code but does not restrict how the surviving code is reached. Machete can be deployed alongside any CFI mechanism: the debloated, phase-restricted binary benefits from both a smaller gadget pool and constrained control-flow transfers.

2.6 Summary and Positioning

Four related gaps emerge from the existing work. First, software debloating reduces mapped code but struggles to balance soundness against aggressiveness; binary-level approaches either sacrifice soundness (RAZOR) or rely on conservative static analysis (Decker). Second, execute-only memory prevents code disclosure but does not reduce the attack surface, since all code pages stay mapped and executable. Third, hardware primitives like MPK allow fast permission switching but need an external policy; existing policy sources are manual annotations, static analysis, or fixed compile-time configurations. Fourth, automata-learning techniques can generalise behavioural models from execution traces, but this has not been applied to derive runtime security policies for memory isolation.

Machete bridges these four gaps in a single system. To the best of our knowledge, it is the first system to combine debloating, execute-only memory, and learned behavioural models into one enforcement framework. The policy-derivation stage learns phase-based memory-access policies from execution traces using a modified Blue-Fringe / EDSM learner, and the runtime stage enforces these policies using process-based page-table isolation, MPK, and SUD, doing so entirely within user space, on unmodified binaries, and without kernel changes.

Compared to the closest prior systems: unlike Decker [7], Machete learns execution phases dynamically rather than computing them statically; unlike TLASR [8], it requires neither kernel extensions nor manual source-level annotations; unlike SMV [9], it derives its policy automatically rather than requiring programmer specification and runs completely in user-space.

Machete: User-Space Temporal Permission Tightening

3

This chapter presents the design and implementation of Machete, a software debloating framework that restricts an application's access to its own executable memory at page granularity. Machete operates entirely in user space, requires no kernel modifications, and enforces phase-specific memory permissions that go beyond what existing mechanisms such as MPK can express, most notably control over the execute permission on individual pages.

We begin by stating the attacker model and the assumptions under which Machete provides its security guarantees. We then give a high-level overview of the system's three-stage pipeline before describing each stage in detail: profiling, policy derivation, and runtime enforcement. The chapter concludes with a discussion of the key design decisions and their limitations.

3.1 System & Attacker Model

Debloating and XoM mechanisms aim to reduce both the permissions and the volume of executable data in memory. To fairly evaluate the security guarantees of our design under realistic worst-case conditions, we therefore adopt a strong attacker model: we assume that the attacker has obtained arbitrary read and write primitives on an already vulnerable application and seeks to escalate to arbitrary code execution using these primitives.

We also make assumptions regarding the binary that we will be debloating. Namely, we assume that the program is a W⊕X binary, meaning there are no regions in the program's memory mappings that are both writeable and executable at once. This is a reasonable assumption, as modern operating systems enforce W⊕X by default through hardware support for the NX (No-Execute) bit [66], and most applications that do not rely on JIT compilation do not map writable and executable memory regions [33]. Under this assumption, our attacker is restricted to reusing existing executable code mapped by our program to build malicious execution chains.

Finally, our execution sandbox leverages the Linux kernel's Syscall User Dispatch mechanism to interpose memory-related system calls and the Memory Protection Keys feature available on recent Intel and AMD CPUs.

We explicitly exclude several classes of attacks from our model. We assume a trusted operating system and kernel, and do not consider compromises of the kernel, hypervisor, or underlying hardware. We do not attempt to defend against side-channel attacks (such as cache or speculative execution side channels), denial-of-service attacks, or adversaries that can modify the program binary on disk before deployment. Finally, we treat applications that rely on just-in-time (JIT) compilation or self-modifying code as out of scope, since they violate the W⊕X assumption on which our design relies.

3.2 System Overview

Machete operates in three stages: profiling, policy derivation, and runtime enforcement. Each stage builds on the artefacts produced by the previous one, and together they form a pipeline that transforms an unmodified application binary into a hardened deployment in which only the memory pages required by the current execution phase are accessible.

In the first stage, the profiler runs the target application under a segmentation-fault–based tracer that records every file-backed page access (read, write, or execute) and every system call, at page granularity. Multi-threaded execution is handled by pinning the application to a single core during profiling and logging per-thread metadata so that access patterns can later be attributed to individual thread levels. In the second stage, the collected execution traces are converted into a prefix tree acceptor (PTA) that captures all observed execution-segment sequences. A Blue-Fringe / EDSM learner then merges states in the PTA to produce a compact deterministic finite-state machine (FSM) whose states represent execution phases and whose transitions are triggered by newly observed page accesses or system calls. For multi-threaded programs, a separate FSM is learned per thread level and the resulting machines are organised into a tree that mirrors the application's thread-spawning hierarchy. In the third stage, the runtime sandbox enforces the learned FSM by mapping each phase to a dedicated operating-system process that shares the application's physical memory but maintains its own page-table permissions. Phase transitions are performed via register-level context switches through shared memory, avoiding the overhead of repeated `mprotect` calls. A lock-step synchronisation protocol keeps virtual memory consistent across all phase processes, and the sandbox itself is hardened with randomised Syscall User Dispatch (SUD) offsets, execute-only mappings, and MPK protection.

The remainder of this chapter details each stage in turn.

3.3 Profiling Memory Access Patterns

As established before, Machete relies on information collected about a program's memory access patterns during a profiling phase in order to calculate a memory-usage policy that is later enforced at runtime. In this section, we describe the steps we take to obtain a comprehensive usage profile for the application under analysis.

Our primary goal in this profiling step is to determine which parts of the virtual address space the application depends on and how different memory regions depend on one another, at page granularity. Because dynamic memory regions such as the stack and heap have unpredictable layouts and cannot contain executable code in a W⊕X program, we focus on virtual memory pages that are backed by files. Concretely, the profiler needs to answer three questions:

- Which virtual memory pages are accessed during the program's lifetime?
- What is the type of each access (read, write, execute)?
- What are the temporal and dependence relationships between these accesses?

We employ a segmentation-fault–based profiling approach. At startup, the profiler disables all permissions on file-backed virtual memory mappings, registers a segmentation fault signal handler, and then re-enables pages on demand while logging all resulting faults, as illustrated in Figure 3.1. To handle permission changes initiated via `mprotect` and `mmap`, the profiler also needs to interpose relevant system calls; we do so using the Linux kernel's SUD mechanism.

Upon startup, the profiler creates a Unix pipe and a separate **watchdog** process that continuously reads from the shared pipe and writes messages to a log file. Once this communication channel is established, the profiler runs an initialisation routine that populates an internal data structure containing all file-backed virtual memory pages in the process's memory map, sends this data structure to the watchdog through the pipe, and sets all file-backed memory pages' permissions (except the interposer) to `NONE`. After registering its segmentation fault handler, it transfers

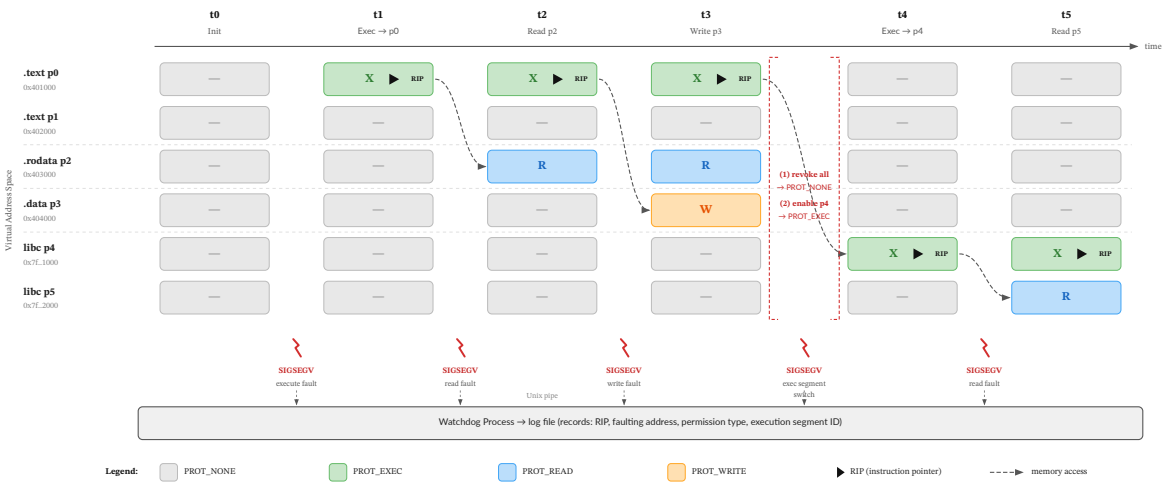


Figure 3.1: Memory permission state during segfault-based profiling. Each column shows the permissions of all file-backed pages after the profiler handles a fault. Every fault is logged to the watchdog process via a Unix pipe.

execution back to the program, which immediately causes a segmentation fault and triggers the handler.

The segmentation fault handler itself contains only a short code sequence that checks whether the fault is valid (that is, not due to an MPK fault or an unmapped page), saves the register state on the stack, and overwrites the return instruction pointer (RIP) so that it points to a dedicated trampoline. This design reduces the risk of encountering a segmentation fault or deadlock inside the signal handler itself, since recovering from either scenario is not possible.

In the trampoline, we must first determine the type of access that caused the segmentation fault: write, read, or execute. Detecting faults due to missing execute permissions is straightforward, because they are the only type of fault where the faulting address is equal to RIP. Special care is needed for instructions at page boundaries; these can be detected by checking whether the instruction at RIP overlaps with the faulting address when the faulting address lies on a page boundary. To distinguish between read and write faults, we inspect the instruction that caused the fault. For this purpose, we use the community-made `iced_x86` library[2] to decode x86 instructions and infer the memory addresses they access.

After determining the type of access that caused the fault, the framework takes one of two paths. For read or write accesses, it enables the corresponding permission on the faulting page and logs the access. For execute accesses, the application undergoes an execution segment switch procedure (see t3→t4 in Figure 3.1): the profiler revokes all permissions from all file-backed virtual memory pages and then re-enables permissions only for the faulting page. This ensures that the profiling stage captures the executable pages from which a given page is accessed within the observed runs and yields a more complete memory-usage profile for those executions.

3.3.1 Ensuring Correctness During Parallel Execution

Additional care is required when applying this segmentation-fault–based profiling technique to multi-threaded programs. Because all threads share the same address space, the profiler might miss

[2]https://docs.rs/iced-x86/1.21.0/iced_x86/

memory accesses made by one thread if the accessed page has already been enabled by another thread. To mitigate this, we reduce parallel execution and encourage interleaved thread scheduling by setting the process's CPU affinity to a single core while bootstrapping the profiler. This increases profiling-time performance overhead but is a necessary trade-off to improve correctness. It does not affect the application's performance once the profile has been obtained, since the affinity restriction is only applied during the profiling phase.

During profiling, we also log thread creation events and append additional metadata to each log entry so that we can later analyse the access patterns of each thread separately.

3.3.2 Detecting Non-Trivial Memory Accesses

The segmentation-fault–oriented profiling technique reliably detects direct memory accesses made in user space. However, it cannot observe indirect memory accesses performed in the kernel when the profiled program issues a system call that requires a page to have a particular permission. To address this limitation, we use the syscall interposition framework K23 [53] to intercept system calls and analyse their arguments to determine whether they require specific page permissions. If the required permission is missing, the profiler logs the access and updates the page's protections accordingly.

3.3.3 Working with Dynamically Mapped Files

Many applications dynamically map files into their virtual address space, for example to load libraries or other resources. The profiling stage must therefore be aware of these dynamically mapped files so that accesses to them can also be detected and logged. To achieve this, we interpose all `mmap` system calls and check whether the requested region is file-backed. If so, we store information about the newly mapped pages in an internal data structure and treat those pages like any other file-backed page, enabling them only on access. This yields a comprehensive usage profile of all file-backed memory over the program's lifetime.

3.4 Deriving Sandbox Policies

Once the profiling stage is complete, Machete produces artefacts in the form of execution traces that record which pages the program accessed, which system calls it issued, and the order in which these events occurred. These traces characterise concrete runs of the application but are typically specific to individual executions and may differ significantly across runs. Moreover, they do not explicitly capture high-level control-flow semantics such as loops.

3.4.1 Modelling Application Behaviour as a Finite State Phase Machine

To reason precisely about application behaviour across multiple executions and to derive sandbox policies that generalise beyond individual traces, we introduce a formal model of execution phases and transitions between them. The model is inferred from execution traces collected during profiling and abstracts the program's behaviour into a deterministic finite-state transition system. Intuitively, each state corresponds to an execution phase (a behavioural mode), and transitions are triggered by newly observed actions that change the set of enabled items.

We first define the alphabet of observable items. Each item represents an operation on an object:

$$i = (\mathrm{Op}, T), \quad i \in \Sigma \tag{3.1}$$

where Op is the operation class (e.g., read, write, execute, syscall), T is the object acted upon (e.g., a virtual page or a system call name), and Σ is the finite set of all observable items. Example items include (Read, 0x1000) or (Syscall, open).

We define phases directly as sets of enabled items:

$$Q \subseteq 2^{\Sigma} \tag{3.2}$$

where Q is the set of phases (states), and each $q \in Q$ is an item-set signature describing which items are enabled or active in that phase.

Transitions between phases are defined over trigger items:

$$\delta : Q \times \Sigma \to Q \tag{3.3}$$

where $\delta(q, a)$ is the successor phase reached from phase q when item a triggers a transition. The transition function is partial: not every pair (q, a) must be defined. To constrain how transitions may occur, we impose a legality condition that captures the "new-item trigger" rule: transitions are only triggered by items that newly appear and must remain enabled in the successor phase. Formally,

$$\delta(q, a) = q' \Rightarrow a \notin q \wedge a \in q' \tag{3.4}$$

meaning that if a transition on item a from phase q to phase q' exists, then a must not be in the set associated with q and must be in the set associated with q'.[3]

We represent a concrete execution as a sequence of observed item sets:

$$\tau = (S_0, S_1, ..., S_n), \quad S_k \subseteq \Sigma \tag{3.5}$$

where each S_k is the set of items observed or enabled at step k of an execution. The sequence (S_k) is produced by profiling and feature extraction over the raw logs to identify the "execution segments" in each raw trace. For two successive observations, we define the set of candidate trigger items as the newly appearing items:

$$\Delta(S_k, S_{k+1}) = S_{k+1} \setminus S_k \tag{3.6}$$

where $\Delta(S_k, S_{k+1})$ is the set of items that appear in S_{k+1} but not in S_k. These items are potential explanations for a phase change between steps k and $k + 1$ and must respect the legality constraint above.

Given a set of execution traces

$$R = \{\tau_1, \tau_2, ..., \tau_m\} \tag{3.7}$$

the learning objective is to infer a phase-transition system

$$M = (Q, \Sigma, \delta, q_0) \tag{3.8}$$

that is consistent with all traces in R. Here, $q_0 \in Q$ is the initial phase. Consistency means that for each trace $\tau \in R$, there exists a corresponding sequence of phases $(q_0, q_1, ..., q_n)$ such that successive phase changes obey the transition function δ.

This model captures the essence of the application's behaviour in the following sense: When the program is in phase q, it is allowed to perform all actions represented by items in q. From this model, obtaining a sandbox policy is straightforward: all pages associated with a phase q are accessible while the program is in that phase, and other pages are not. As a result, the program behaves as expected during the given phase. When it attempts to access a page that should trigger a phase transition, the access causes a segmentation fault. Control is transferred to the sandbox, which compares the observed operation against the candidate trigger items on the outgoing transitions of the current phase. If the operation matches one of those items, the sandbox takes the corresponding transition, enables the necessary pages, and resumes execution. The set-valued trigger domains

[3]This condition can also be partially captured by modelling each item $a \in q$ as a self-loop transition $\delta(q, a) = q$. However, using this notation makes the FSM-learning stage more complex.

introduced during learning arise from segment-level aggregation in the traces, but at runtime the sandbox still observes and matches one concrete page access or system call at a time. Note that any transition items i are disabled during a phase due to the legality constraint (i.e., $\delta(q, a) = q' \Rightarrow a \notin q$). It must also be noted that transition items with Op = Syscall are directly observable by the sandbox using system call interposition.

3.4.2 Building a Prefix Tree Acceptor From Execution Traces

Since Machete's raw output is a set of execution traces at the level of individual memory and syscall events, the first step is to combine them into a single structure that can be refined into a more compact FSM. To this end, we build a prefix tree acceptor (PTA) from the execution traces [59]. The PTA is a deterministic finite-state transition system that accepts exactly the observed traces.

The atomic units when processing an execution trace are the execution segments that the application has entered during its runtime. This is because the profiler does not log how many times a page access happened, only its first occurrence, meaning we cannot reliably order items within an execution segment. Segments can be reconstructed from the raw logs by splitting each trace at execute events and grouping all items between two successive execute events (including the execute event itself) into one segment. Formally, an execution trace is represented as a sequence of execution segments $\tau = (S_0, S_1, ..., S_n)$, where each $S_k \subseteq \Sigma$ is a set of items.

In the trivial case with a single execution trace $\tau = (S_0, ..., S_n)$, the PTA is a trie whose nodes correspond to execution segments and whose edges represent candidate trigger sets between segments. The root node represents the initial phase q_0 with item-set signature $\varnothing$. For each successive pair (S_k, S_{k+1}), we define the edge label as the set of newly appearing items: $\Delta(S_k, S_{k+1}) = S_{k+1} \setminus S_k$. Intuitively, an edge from a phase with item set S_k to a phase with item set S_{k+1} is annotated with the candidate trigger domain $\Delta(S_k, S_{k+1})$.

Although representing an edge label as a set of items is not conventional, it makes the PTA construction more intuitive: a candidate trigger set $D = \{i_1, ..., i_\ell\}$ can equivalently be interpreted as multiple single-item transitions (q, i_j, q') for each $i_j \in D$. We keep the set-based view, because it directly mirrors $\Delta(S_k, S_{k+1})$ and simplifies the later learning stage.

To construct a PTA that combines multiple execution traces, we incrementally insert each trace while reusing and generalising previously created phases. In this construction, phases are represented directly as item sets, so each phase variable (e.g., q) denotes its own signature. Concretely, for each trace $\tau = (S_0, ..., S_n)$ we traverse from the root phase q_0 and, at each segment S_k, compute the set of newly appearing items $\Delta_k = S_k \setminus S_{k-1}$, with the convention $S_{-1} := \varnothing$. If $|\Delta_k| = 0$, the current segment does not introduce any new items, so we simply advance the segment baseline by setting $S_{k-1} := S_k$ and continue.

When $|\Delta_k| > 0$, we attempt to follow an existing outgoing edge whose candidate trigger set is compatible with Δ_k. We distinguish three cases for an outgoing edge from the current phase with candidate domain D: (i) $\Delta_k \subseteq D$, in which case we can follow this edge without structural changes because all newly appearing items in this segment are already covered by the existing candidate domain; and (ii) $D \subseteq \Delta_k$, in which case we still follow the edge but grow it by enlarging the target phase's item-set signature and the edge's candidate domain to also include Δ_k. If no outgoing edge is compatible with Δ_k (neither $\Delta_k \subseteq D$ nor $D \subseteq \Delta_k$ for any outgoing edge domain D), we create a fresh phase with item-set signature S_k and a new edge from the current phase labelled with Δ_k. The complete construction is shown in Algorithm 3.1.

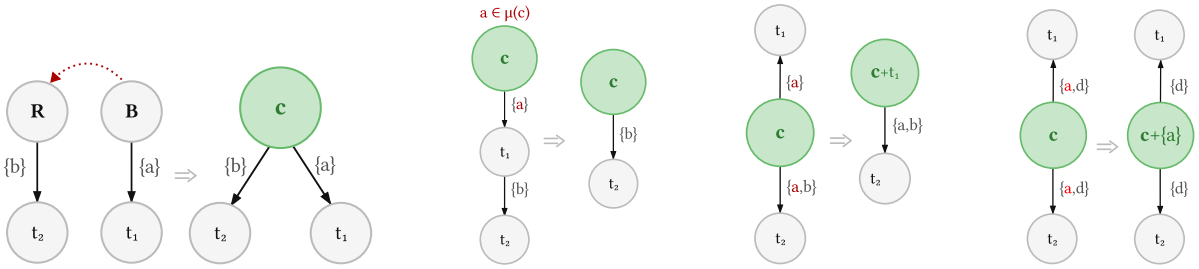
Algorithm 3.1: PTA construction and shared-item lifting procedures

procedure BUILD-PROTO-PTA(R, Σ)

$q_0 \coloneqq \emptyset$; $Q_E \coloneqq \{q_0\}$; $E_E \coloneqq \emptyset$
for each trace $\tau = (S_0, ..., S_n) \in R$ **do**
  $q \coloneqq q_0$
  $S_{-1} \coloneqq \emptyset$
  for $k = 0, 1, ..., n$ **do**
    $S \coloneqq S_k$
    $\Delta_k \coloneqq S \setminus S_{k-1}$
    if $\Delta_k = \emptyset$ **then**
      $S_{k-1} \coloneqq S_k$
      continue
    else
      if exists $(q, D_{\text{match}}, q') \in E_E$ **with** $\Delta_k \subseteq D_{\text{match}}$ **then**
        $q \coloneqq q'$
      else if exists $(q, D_{\text{match}}, q') \in E_E$ **with** $D_{\text{match}} \subseteq \Delta_k$ **then**
        $N \coloneqq S \setminus q'$
        $q' \coloneqq q' \cup S$
        extend D_{match} with all items from Δ_k
        $(q, E_E) :=$ LIFT-SHARED-ITEMS(q, E_E, Sigma)
        for each outgoing edge from q' with domain D_{out} and target q'' in E_E **do**
          remove all items in N from D_{out}
        $q \coloneqq q'$
      else
        let $q_n \coloneqq S$
        $Q_E \coloneqq Q_E \cup \{q_n\}$
        $E_E \coloneqq E_E \cup \{(q, \Delta_k, q_n)\}$
        $(q, E_E) :=$ LIFT-SHARED-ITEMS(q, E_E, Sigma)
        $q \coloneqq q_n$
      $S_{k-1} \coloneqq S_k$
return $M_E = (Q_E, E_E, q_0)$

procedure LIFT-SHARED-ITEMS(q, E_E, Σ)

let $(q, D_1, q_1), ..., (q, D_r, q_r)$ be all outgoing edges from q in E_E
let L be the set of items in Σ that appear in at least two of $D_1, ..., D_r$
if $L = \emptyset$ **then return** (q, E_E)
$\hat{q} \coloneqq q \cup L$
$\hat{E} \coloneqq E_E$
for each outgoing edge $\left(q, D_j, q_j\right)$ in $E_\wedge$ **do**
  $D_j \coloneqq D_j \setminus L$
return $\left(\hat{q}, \hat{E}\right)$



(a) Clean merge (b) Absorption (c) Cascade (d) Item lifting

Figure 3.2: Stabilisation cases during Blue-Fringe merging. Green nodes denote the merged class c; grey nodes denote cascade-merged targets. ⇒ denotes the merge operation.

We enforce determinism of the PTA with respect to candidate triggers using the helper routine at Line 3.31. Given a phase q, the routine computes the set of shared candidate items and returns an updated pair $(\hat{q}, \hat{E})$ in which shared items have been moved into the phase set and removed from outgoing edge domains. This guarantees that no item can appear as a candidate trigger on two sibling edges, which preserves determinism.

When items are lifted into a target phase q' (for example during edge growth), we also propagate this change to its outgoing edges: items that have become part of the phase set q' are removed from the candidate domains of all edges leaving q'. This keeps a consistent separation between items that are always enabled in a phase and items that serve as candidate triggers for leaving that phase.

3.4.3 Blue-Fringe FSM Learning

Starting from the proto-PTA $M_E = (Q_E, E_E, q_0)$, the objective of Blue-Fringe learning [59,60] is to infer a compact deterministic model $M_L = (Q_L, \Sigma, \delta_L, q_0^L)$ that remains consistent with the observed executions while generalising beyond individual runs. This step is necessary because the raw PTA can be large and highly trace-specific, which makes direct policy derivation difficult and brittle in practice.

We use a modified Blue-Fringe / EDSM-style learner because it provides a practical balance between structural guarantees and search efficiency: RED/BLUE frontier control bounds the merge search space while evidence-driven scoring [67] in turn prioritises merges that simplify the machine without collapsing semantically important distinctions. This is particularly useful for our setting, where traces are rich in low-level events and naive global merging is both expensive and unstable.

We maintain a hypothesis $H_t = (\Pi_t, \mu_t, F_t)$ at iteration t. Here, Π_t is the current equivalence relation over proto-phases; equivalently, it induces a set of equivalence classes C_t, where each class represents one candidate learned state. For any class $c \in C_t$, $\mu_t(c) \subseteq \Sigma$ denotes its enabled-item signature, and $F_t(c)$ denotes its outgoing frontier with refined trigger domains. RED, BLUE, and WHITE are subsets of C_t: RED is the committed core, BLUE is the immediate successor frontier of RED, and WHITE contains the remaining classes. For each selected $b \in B_t$, we compute a pre-score against every RED candidate $r \in R_t$, sort RED candidates by that score, and then evaluate them in that order with full tentative merges.

For a tentative merge between RED class r and BLUE class b, we first form a merged class $c = r \cup b$, set $\mu_t(c) = \mu_t(r) \cup \mu_t(b)$, and combine both outgoing frontiers into $F_t(c)$. We then stabilise the

result case by case. If refinement removes only already-enabled items, no structural repair is needed (Figure 3.2.a). If an outgoing domain becomes empty, the corresponding target class is absorbed into c (Figure 3.2.b). If overlapping target groups create subset conflicts, cascade merges are applied until consistency is restored (Figure 3.2.c). If sibling transitions share trigger items, shared items are lifted into the source signature and removed from sibling domains to maintain deterministic outgoing behavior (Figure 3.2.d). Stabilisation iterates until no further absorptions, cascades, or domain updates occur.

The post-score is computed only after this fixed point is reached, so it captures both direct and induced effects of the candidate. The acceptance policy is greedy: scanning RED candidates in pre-score order, we accept the first candidate with post-score at least θ. If no RED candidate satisfies the threshold, the BLUE class is promoted to RED. This preserves the Blue-Fringe frontier invariant and guarantees monotone progress until $B_t = \emptyset$.

The learned FSM translates directly into a runtime sandbox policy. Each state corresponds to a phase in which a fixed set of pages is accessible; all other file-backed pages are unmapped or marked inaccessible. When the program attempts an access that is not permitted in the current phase or tries to make a syscall, the action hands control to the sandbox, which compares the observed event against the candidate trigger items associated with the outgoing transitions of the current phase. If a matching transition exists, the sandbox switches to the target phase by updating page permissions accordingly; otherwise, the access is treated as a policy violation. The compactness of the FSM therefore determines both the precision of per-phase permissions and the frequency of runtime transitions.

3.4.3.1 Scoring Merges

The scoring procedure is split into a ranking phase and a decision phase. The ranking phase uses a lightweight *pre-score* to order RED candidates for a fixed BLUE class before any stabilisation work is performed. The decision phase computes a *post-score* after full stabilisation and applies the greedy threshold rule described above.

Both scores are weighted linear combinations of feature terms that capture different aspects of merge quality. The pre-score evaluates candidates based on four criteria: (i) the overlap between the enabled-item signatures of the RED and BLUE classes, measured by their Jaccard similarity; (ii) a penalty for mismatches between the types of items present in each signature; (iii) a reward for structural simplification, favouring candidates whose merge would collapse transition complexity; and (iv) a penalty that discourages merging classes whose syscall-bearing transitions would be lost, since these transitions are cheap to enforce at runtime and serve as useful phase discriminators.

The post-score is computed after the stabilisation fixed point (Line 3.20) and therefore accounts for cascade effects that are invisible to the pre-score. It evaluates: (i) the same signature overlap as the pre-score; (ii) a penalty proportional to the number of cascade merges and absorptions triggered during stabilisation; (iii) a penalty for items removed from transition domains as a side effect of stabilisation; (iv) a penalty for the total domain size of cascade-merged edges, which quantifies how much frontier structure was disrupted; (v) the net change in structural transition cost; and (vi) the net change in syscall transition cost. A candidate is accepted if its post-score meets the threshold θ.

In practice, θ is chosen empirically during calibration on representative profiling traces, guided by the trade-off between model compactness and phase precision.

Adjusting the scoring coefficients directly controls the security–performance trade-off: higher cascade and disruption penalties preserve fine-grained phases with tighter permissions but more

frequent runtime transitions, while a larger structural-simplification reward merges phases aggressively, reducing overhead at the cost of wider per-phase permission windows. For instance, increasing the penalty for merging phases whose execute-item signatures differ produces more phases, each with a narrower set of executable pages, at the expense of additional phase transitions and therefore higher runtime overhead. The precise scoring formulas and feature definitions are given in Appendix A.

3.4.4 Building FSMs For Multi-Threaded Programs

The Blue-Fringe learner described above operates on a single set of execution traces that share a common thread of control. To handle multi-threaded programs, we decompose the learning problem hierarchically by *thread level*: a hierarchical identifier that distinguishes the main thread from each child thread it spawns, and their children recursively. The profiler records, for each execution segment, the thread level at which it was observed and annotates thread-spawning events with the level of the newly created thread. Given a set of execution traces, we first partition all segments by thread level and learn an FSM exclusively from the root-level segments using the procedure described in the preceding sections.

Once the root-level FSM has been learned, we inspect each of its states for segments that contain thread-spawning events. For every child thread level discovered in this way, we collect the corresponding segments across all traces and submit them as a new, independent learning job. This process recurses: each child FSM is itself inspected for further spawns. The result is a tree of FSMs in which each node contains a learned phase machine for one thread level and edges record which parent state triggered the spawn. This decomposition assumes that the thread-spawning structure and the dominant per-thread access patterns are sufficiently stable across the profiled runs for one FSM per thread level to remain meaningful. At runtime, the sandbox maintains one active phase per live thread level and enforces each thread's transitions independently, using the parent–child links to activate the appropriate child FSM when a thread-spawning event is observed.

Figure 3.3 shows two representative FSMs produced by this pipeline for different benchmark applications.

3.5 Enforcing the Sandbox During Runtime

The previous sections described how Machete profiles an application's memory access patterns and derives a finite-state phase machine whose states encode per-phase permission sets. In this section, we describe how the sandbox enforces these permissions at runtime. The central challenge is to switch the set of accessible pages on every phase transition with minimal overhead, while preserving the correctness and consistency of the application's virtual address space.

3.5.1 Per-Phase Memory Views

A naive enforcement strategy would call `mprotect` on every affected page at each phase transition. For phases that differ in the permissions of many pages, this approach incurs substantial overhead due to the large number of system calls required [7,41]. To avoid this cost, we instead represent each phase as a separate operating-system process with its own virtual address space and page tables. During bootstrapping, we remap the application's memory regions as shared between all phase processes. Because each process maintains independent page-table entries for the same underlying physical pages, each phase can hold distinct permissions for the same data in memory.

Algorithm 3.2: Blue-Fringe / EDSM learning loop.

procedure LEARN-BLUE-FRINGE(M_E, θ)

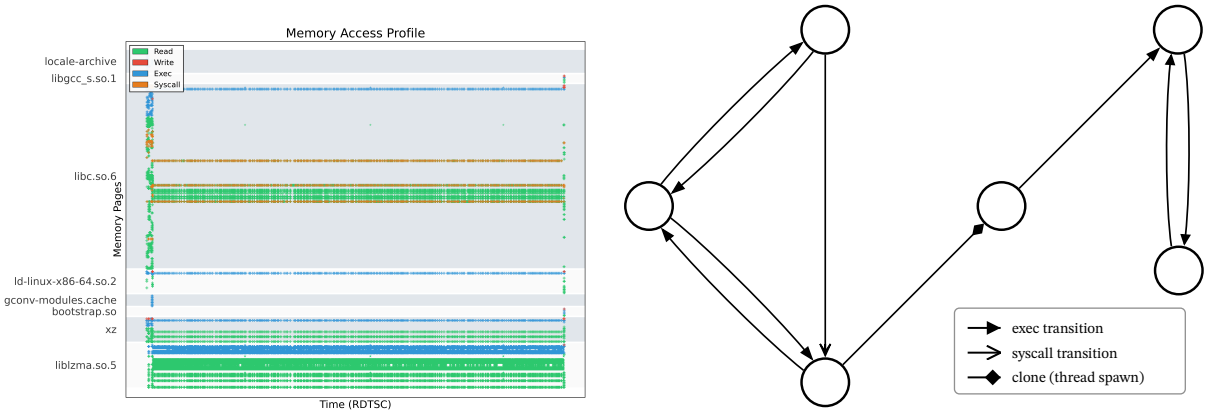
$H \coloneqq$ hypothesis induced by M_E
$R \coloneqq \{q_0\}$; $B \coloneqq$ non-RED successors of R in H
while $B \neq \varnothing$ **do**
  choose $b \in B$
  for each $r \in R$ **do** compute $s_{\text{pre}}(r, b)$
  let $\mathcal{R} \coloneqq R$ sorted by descending s_{pre}
  accepted $\coloneqq$ false
  for each $r \in \mathcal{R}$ **do**
    let $H' \coloneqq H$ ▷ snapshot current hypothesis
    $s \coloneqq$ TRY-MERGE-AND-STABILISE(H, r, b)
    if $s_{\text{post}}(r, b; s) \geq \theta$ **then**
      LIFT-SHARED-ITEMS at representative of r in H
      accepted $\coloneqq$ true
      break
    else $H \coloneqq H'$ ▷ restore hypothesis
  if ¬ accepted **then** $R \coloneqq R \cup \{b\}$
  $B \coloneqq$ non-RED successors of R in H
return learned FSM induced by H

procedure TRY-MERGE-AND-STABILISE(H, r, b)

merge representatives of r and b; let merged representative be c
$W \coloneqq [c]$; initialise merge statistics s
while $W \neq \varnothing$ **do**
  pop state x from W
  refine outgoing domains of x by removing items in $\mu(x)$
  if exists empty-domain edge from x to target t **then**
    absorb t into x; update s; push x to W; **continue**
  if exists overlapping target groups with subset conflict at x **then**
    cascade-merge conflicting targets; update s
    push affected representatives to W; **continue**
  lift shared sibling trigger items at x into $\mu(x)$; update s
return s

This approach emulates the per-phase isolation achieved by kernel-level mechanisms such as SMV [9], but operates entirely in user space and requires no kernel modifications.

Since the learned FSM is available at startup, the sandbox can pre-configure the appropriate page permissions for every process during the bootstrapping stage. Once this initialisation is complete, a phase transition reduces to transferring the execution state from one process to another



(a) Memory Access Behaviour of xz (b) State Machine Learned From the Behaviour

Figure 3.3: The memory access pattern of xz over time, where each dot represents a memory event on a page logged during profiling. The resulting state machine is a compact representation of the memory access patterns, where each state represents a different phase of the application's execution.

and allowing the target process to continue where the previous one left off, without any per-page permission changes at transition time.

This design offers two additional advantages beyond reduced overhead. First, it allows multiple logical threads to execute concurrently, each within a phase that maintains a distinct set of memory permissions, without the permission changes of one thread interfering with those of another. Achieving this with `mprotect` alone is not possible since `mprotect` operates on the shared address space of the entire. Second, although Memory Protection Keys (MPK) can provide similar per-thread permission domains with even lower overhead, MPK does not support controlling the execute permission on a page [66], which is one of the main contributions of Machete.

3.5.2 Guaranteeing Virtual Memory Consistency Across Phases

Although existing pages are remapped as shared during startup, most off-the-shelf binaries rely on dynamically loaded shared libraries or map entirely new memory regions at runtime. Because each phase process has an independent virtual address space, operations that affect virtual memory in one process do not automatically propagate to the others. This requires careful handling to ensure that all phase processes maintain a consistent view of the application's address space throughout execution.

We address this problem by interposing all system calls related to virtual memory management and synchronising their effects across all phase processes. To intercept these calls, we reuse the Syscall User Dispatch (SUD)–based interposition mechanism employed during profiling. Each phase process runs a dedicated synchronisation thread in the background that listens on a shared lock-free queue for incoming synchronisation requests. The queue is processed in lock-step order, ensuring that all memory operations are applied in the same sequence across all phases.

When a process intercepts a virtual-memory system call, it pushes the corresponding operation onto the shared queue and waits for all synchronisation threads to reach the same point. Each synchronisation thread, upon reaching the pending operation, waits on a shared futex to receive the result. Once all synchronisers are ready, the originating process acts as the leader: it executes

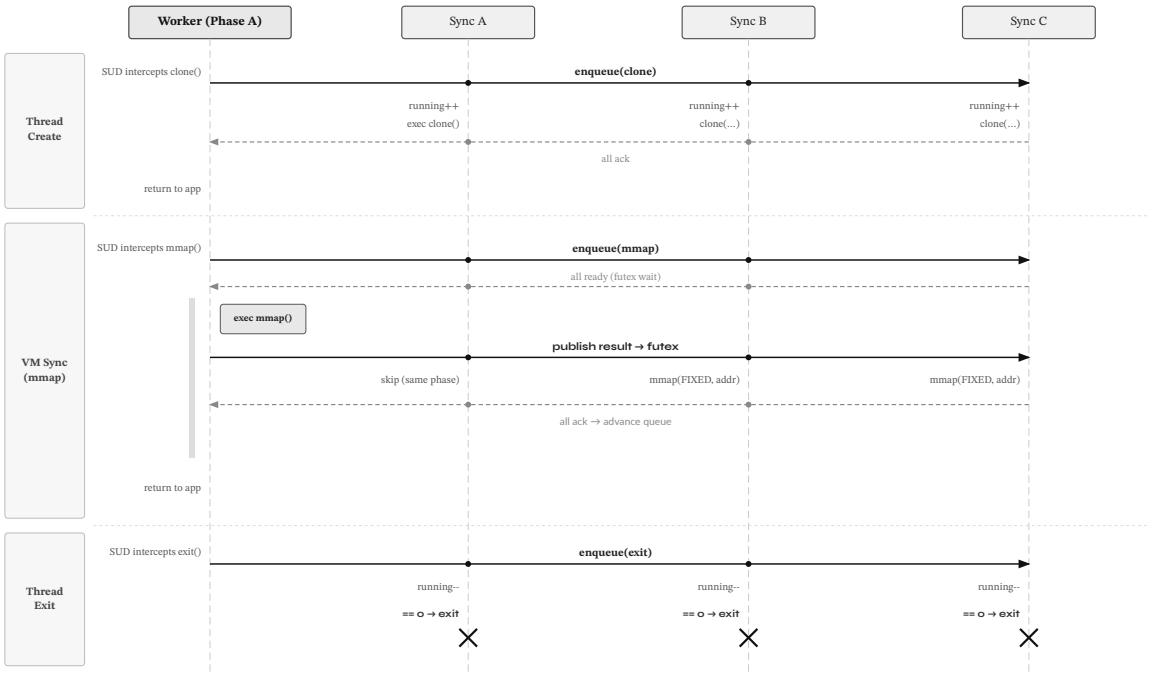


Figure 3.4: VM consistency protocol. The leader process (A) intercepts a virtual-memory system call, enqueues the operation, and waits for all follower processes (B, C) to reach the barrier. The leader executes the syscall and publishes the result; followers apply the operation locally before acknowledging completion.

the system call and writes the result to the futex. The remaining phase processes use this result as a source of truth when performing the operation in their own address spaces. For example, when a new page is mapped in one phase, the synchronisation threads in the other phases map the same page at the fixed address published by the leader. The originating process is permitted to return from the system call only after all synchronisers have completed the operation and advanced to the next queue entry. This protocol ensures that a thread never encounters a situation in which its current and next phase hold different virtual memory states. Figure 3.4 illustrates the lock-step synchronization protocol used to maintain consistent virtual memory state across all phase processes.

3.5.3 Transferring Control Across Phases

The process-based phase model described above effectively achieves a form of user-space virtualisation that simulates a continuous execution flow across the application's lifetime. In practice, the application constantly transitions between different processes while preserving its control flow intact. From Machete's perspective, a *thread* therefore corresponds to a logical control-flow entity rather than a literal operating-system thread.

Because the runtime guarantees virtual memory consistency across all phase processes, transferring execution from one phase to another reduces to a register-level context switch. When a phase transition is triggered, the sandbox snapshots the complete register state of the active thread, including general-purpose, stack pointer, instruction pointer, extended (e.g., SSE/AVX), and thread-local storage registers, and writes this snapshot to a shared memory region. The target phase's process is then woken, restores the saved register state, and resumes execution from the exact point where the previous phase left off.

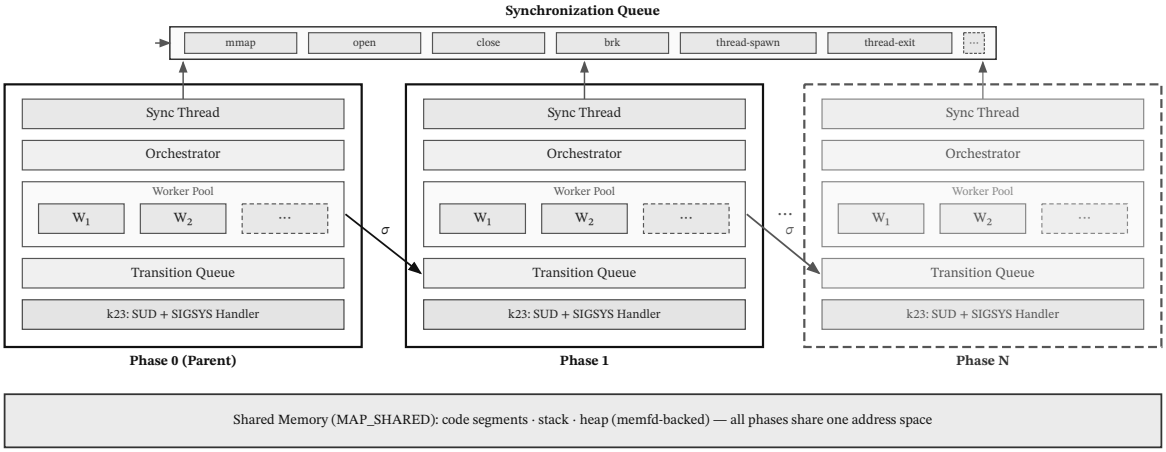


Figure 3.5: Multi-process phase architecture. Each phase runs in a separate process with its own page-table permissions over the same shared physical memory. Workers push their state σ to the target's transition queue.

A single FSM state may, however, host multiple concurrent logical threads. Dedicating the entire phase process to a single execution flow would block other threads from making progress. To address this, we model each phase process as an orchestrated container that manages multiple *worker* threads internally. Each worker thread shares its virtual memory with the parent phase process and is responsible for executing a single logical thread. Workers listen on a shared lock-free queue for incoming transition events directed at their phase. When a transition event is enqueued, a pending worker wakes up, restores the incoming register state, and begins executing the logical thread. When the latter reaches a subsequent phase transition, the worker pushes an outgoing event onto the target phase's queue and returns to listening for the next event.

Each phase has an additional *orchestrator* thread that monitors the amount of free worker threads and spawns new ones to prevent worker starvation. Pre-emptive spawning is preferred over on-demand creation because each new worker thread must initialise its SUD configuration and segmentation fault handler, which would otherwise introduce additional latency on the critical path. Figure 3.5 illustrates the resulting multi-process architecture.

3.5.4 Hardening the Execution Environment

The process-based segregation model with shared data structures for efficient control-flow transfers is a powerful enforcement technique, but it also introduces an additional attack surface that must be protected. We take several measures to harden the debloating runtime against exploitation.

First, for system call interposition, we use an enhanced version of the K23 framework [53] with additional measures to prevent bypasses. At startup, the sandbox collects all system call sites recorded during profiling and rewrites their `syscall` instructions with `call *%rax` instructions that redirect to a trampoline mapped at address `0x0`. For any system call sites that the profiling stage did not observe, SUD remains enabled as a fallback interposition mechanism. SUD is controlled by a single byte in the process's memory, located at an offset relative to the GS base of the current thread. To make it harder for an attacker to tamper with this control byte, we do not use the immediate GS base address; instead, all SUD control-value accesses are performed through the GS base plus a constant offset. The latter is hard coded into the binary and is re-randomised each time the

```
movabs rax, 0xdeadbeefcafebabe ; Randomized offset
mov gs:[rax], 1                ; Disable SUD
xor rax, rax                   ; Clobber to prevent disclosure
```

Listing 3.1: SUD control byte toggle with randomised GS-base offset. The offset is re-randomised at each launch and clobbered after use.

application is launched. The sandbox code itself is mapped as execute-only, preventing the attacker from reading the randomised offset. Listing 3.1 illustrates this rewriting scheme.

Second, we rewrote the K23 framework so that the sandbox is fully self-contained and does not depend on any shared libraries. This ensures that the sandbox itself does not introduce spurious memory-access entries during profiling and is not affected by the debloating of library pages at runtime.

Third, the entire address space belonging to the sandbox, including its heap, signal stack, and all memory pages used for inter-phase communication, is protected using MPK. This prevents the application from reading or writing sandbox-internal data structures, even if an attacker has obtained arbitrary read and write primitives within the application's own address space.

3.6 Design Decisions and Limitations

The defining architectural decision in Machete is the use of one operating-system process per FSM state rather than repeated `mprotect` calls or a custom kernel module. This design yields three concrete benefits: it enables control over the execute permission on individual pages, which Memory Protection Keys (MPK) cannot express [66]; it eliminates per-transition system-call overhead by reducing phase switches to register-level context transfers; and it operates entirely in user space, requiring no kernel modifications. The cost is a fixed memory overhead proportional to the number of learned phases and is incurred by both duplicate page-table entries and per-phase synchronisation threads, together with the lock-step virtual-memory consistency protocol described in the preceding section. Because permissions are managed at page granularity, functions that share a page are treated as an indivisible unit; achieving sub-page precision would require binary rewriting or relinking, which we consider orthogonal to this work.

The security guarantees that Machete provides are bounded by the completeness of the execution traces collected during profiling. If a legitimate code path is never exercised, the learned FSM will not contain a corresponding phase, and the runtime will treat the resulting access as a policy violation. Blue-Fringe learning mitigates this limitation by generalising across multiple traces, but it compresses observed behaviour rather than extrapolating to unobserved paths. Similarly, pinning multi-threaded applications to a single core during profiling improves coverage of interleaved thread schedules (albeit at the cost of profiling-time performance), yet it may still miss access patterns that manifest only under true parallel contention. Additionally, the scoring coefficients that govern the security-versus-overhead trade-off are explicitly tunable, but require empirical calibration for each workload class.

Finally, Machete's three-stage pipeline is deliberately modular: each stage is detached from another via artefacts that can be obtained from other sources or used in different runtimes. The learned FSM is a standalone policy specification that can drive enforcement runtimes other than Machete's own sandbox, including kernel-level memory-view mechanisms such as SMV [9] or the mmview-based approach of TLASR [8]. Conversely, the process-based enforcement stage can accept

policies derived by other means, such as the static analysis approach of Decker [7] or compiler annotations. This composability is particularly valuable because Machete is, to the best of our knowledge, the first debloating system that performs temporal debloating over executable pages entirely in user space. Thanks to this, its policy-derivation stage can be integrated into compiler toolchains alongside complementary techniques such as PartiSan [68] to produce hardened binaries that combine phase-gated memory permissions, without requiring kernel modifications.

4 Evaluation

This chapter evaluates Machete in terms of correctness, security, runtime overhead, and sensitivity to sandbox-generation parameters, comparing it to Razor [6] and Decker [7]. Unlike these systems, Machete combines dynamically learned execution phases, page-level enforcement, execute-only memory, and shared-library debloating. After describing the experimental setup, we present a comparative analysis demonstrating that this design achieves substantially stronger attack-surface reduction than static rewriting or function-level temporal partitioning.

4.1 Experimental Setup

We evaluate all three tools on a shared set of 10 core utilities and one server program for which comparable artifacts exist across Razor, Decker, and Machete. The 10 core utilities – bzip2, chown, date, grep, gzip, mkdir, rm, sort, tar, and uniq – originate from the ChiselBench suite [20] and were subsequently adopted by both Razor and Decker as their primary evaluation targets. These programs were originally selected because they are open-source, widely deployed, cover a range of functionality (compression, filesystem manipulation, text processing), and each contains known CVE-listed vulnerabilities [20]. We additionally include lighttpd 1.4.82 that is an HTTP server we debloated with both Razor and Decker, which exercises long-running, server behaviour absent from the utility benchmarks.

Beyond the shared set, we include two Machete-only auxiliary targets: memcached 1.6.19 and xz 5.8.2. These programs are not used for cross-tool comparison but serve to demonstrate Machete's applicability to multi-threaded software outside the shared benchmark set. Table 4.1 summarises the full target list and indicates which tools cover each program.

Each tool works on different binary variants with different security and performance properties by design. For Razor, we compare a non-PIE baseline binary against the debloated binary produced by Razor's stitching and merging pipeline. For Decker, we compare an LLVM compiled PIE baseline binary against its whole-program-debloated (WPD) counterpart; because the WPD binary dynamically loads a runtime support library, we additionally report a WPD without runtime variant that excludes the runtime library's pages from the security accounting. Machete operates on a PIE enabled binary compiled with GCC on the host machine, we compare the baseline configuration (all pages enabled) against the sandboxed binary. We report both an enforced variant (excluding the interpreter library) and an enforced with runtime variant that includes the pages of `libmachete_interp.so` in the security metrics. For lighttpd, we evaluate machete with two sandbox granularities: the default 2-phase (fast) sandbox and a 3-phase (slow) sandbox, allowing us to examine the security–overhead trade-off at different policy granularities.

Finally, we include k23 [53], the syscall interposition framework underlying Machete, as a performance baseline. In this configuration, k23 runs the application in a single process with syscall interposition only: it intercepts every system call but does not perform multi-process phase switching, enforce per-phase page permissions, or apply debloating. Including k23 isolates the overhead of syscall interposition from the additional mechanisms introduced by Machete (segfault-based transition detection, cross-process context switching, and the virtual memory consistency protocol). For this reason, k23 is reported only in the performance results.

Table 4.1: Evaluation targets and tool coverage.

Program	Type	Razor	Decker	Machete	Notes
bzip2 1.0.8	Utility	✓	✓	✓	Compression
chown 8.16	Utility	✓	✓	✓	Filesystem metadata
date 8.16	Utility	✓	✓	✓	Date formatting
grep 2.16	Utility	✓	✓	✓	Pattern matching
gzip 1.2.4	Utility	✓	✓	✓	Compression
mkdir 8.16	Utility	✓	✓	✓	Filesystem creation
rm 8.16	Utility	✓	✓	✓	Filesystem deletion
sort 8.16	Utility	✓	✓	✓	Text sorting
tar 1.28	Utility	✓	✓	✓	Archive manipulation
uniq 8.16	Utility	✓	✓	✓	Deduplication
lighttpd 1.4.82	Server	✓	✓	✓	HTTP server
memcached 1.6.19	Server	×	×	✓	Key-value cache
xz 5.8.2	Utility	×	×	✓	LZMA compression

A caveat that must be noted is that the three tools cannot operate on the same physical binary. Razor requires a non-PIE binary; Decker requires a PIE binary compiled with LLVM 10 and its own build infrastructure; Machete operates on unmodified binaries built with GCC on the host. As a result, the baseline binary differs across tools, and absolute metric values such as total gadget counts or performance are not directly comparable across tool families. Where cross-tool comparison is done, we report baseline values for reference and percentage change relative to each tool's own baseline for a fair comparison.

Absolute counts nevertheless remain meaningful for reasoning about the attack surface at runtime. An attacker who has obtained a memory-corruption primitive operates on the process's virtual address space as it exists at the moment of exploitation: the set of pages currently mapped executable, and the gadgets reachable within them, fully determines the code-reuse material available to the attacker. Build provenance is invisible at exploitation time. Hence, a tool that leaves 50 executable pages in a given phase presents a strictly smaller attack surface than one that leaves 400, even if the two tools started from different baseline binaries.

A related concern is dynamic dependencies and runtime libraries. Mainly, Machete is the only tool out of the three that debloats the entire dependency chain of a program instead of just the binary itself, which means it can achieve significantly better security improvements. Besides the existing shared libraries the programs depend on, both Decker and Machete rely on runtime support libraries that are mapped into the application's address space during enforcement. Decker loads `libdebloat_rt.so` (244 executable pages) and pulls in `libm.so.6` (151 pages); Machete loads `libmachete_interp.so` (82 executable pages). These libraries inflate the executable footprint and must be reported transparently. We therefore present security metrics in both *with runtime* and *without runtime* views throughout this chapter. A deliberate design goal of Machete was to keep its runtime library lightweight: at 82 pages, it is roughly one-fifth the size of Decker's runtime, mitigating the paradox of introducing additional code while debloating.

Table 4.2: Metric categories collected per target and variant.

Category	Metric	Definition
Pages	exec_pages	Number of executable pages available in a phase
	rx_pages	Pages simultaneously readable and executable (R+X)
Gadgets	gadget_count	Total gadgets across all classes (ROP, JOP, COP, syscall)
	rop / jop / cop / syscall	Per-class classified gadget counts
Expressivity	practical_rop_classes	Practical ROP classes satisfied (out of 11)
	aslr_proof_classes	ASLR-proof gadget classes satisfied (out of 35)
	turing_complete_classes	Turing-complete gadget classes satisfied (out of 17)
	sp_gadget_types	Special-purpose gadget types present (out of 10)
Performance	runtime	Wall-clock time (compression, xz): hyperfine, 10–20 runs
	throughput	Requests/sec (lighttpd) or ops/sec (memcached): wrk / memtier

We collect security metrics at per-phase granularity using ROPgadget for gadget enumeration. We also measure the Gadget Set Analytics (GSA) metrics from Brown and Pande [15,69] for classification, as gadget count alone can be a misleading security metric for debloating tools. For each phase of every sandbox, we record the number of executable pages, the number of simultaneously readable-and-executable (R+X) pages, and the classified counts of ROP, JOP, COP, and syscall gadgets. Because Razor has no temporal debloating, we treat it as a single-phase sandbox. We further compute expressivity indicators: practical ROP classes satisfied (out of 11), ASLR-proof classes (out of 35), Turing-complete classes (out of 17), and special-purpose gadget types present (out of 10).

For performance, we measure compression runtime (bzip2, gzip, xz) using hyperfine with 20 runs and 3 warmup iterations, server throughput for lighttpd using wrk (10 runs, 4 threads, 100 connections, 10 s per run), and server throughput for memcached using memtier_benchmark (10 runs, 30 s, 4 threads, 50 clients per thread). Table 4.2 lists the full set of collected metrics.

All experiments were conducted on a single machine running Arch Linux with kernel 6.19.9, equipped with an Intel Core i9-9980XE (18 cores, 3.00 GHz base) and 128 GB of DDR4-2133 MhZ RAM. For all performance measurements, the CPU governor was set to `performance`, turbo boost was disabled, and the page cache was dropped before each measurement session to reduce variance.

4.2 Correctness Analysis

Security and performance claims are meaningful only if the debloated artifacts preserve their intended functionality. Before examining attack-surface metrics or runtime overhead, we therefore verify that each tool's output remains correct on a shared test suite.

The test suite comprises 208 test cases distributed across the 10 shared core utilities. The cases exercise file-corpus inputs, flag variations, pattern-matching workloads, and filesystem operations. Each test compares the debloated program's output and exit code against the original binary; any deviation, timeout, or crash is recorded as a failure. Table 4.3 reports the per-program results for all three tools.

Table 4.3: Per-program correctness results (pass/total). Bold entries indicate failures.

Program	Tests	Razor	Decker	Machete
bzip2	33	33/33	33/33	33/33
chown	3	**2/3**	3/3	**0/3**
date	10	10/10	10/10	**9/10**
grep	20	20/20	20/20	**7/20**
gzip	33	33/33	33/33	**31/33**
mkdir	4	4/4	4/4	**3/4**
rm	4	**3/4**	4/4	4/4
sort	28	28/28	28/28	28/28
tar	33	**32/33**	33/33	33/33
uniq	40	40/40	40/40	40/40
Total	**208**	**205/208**	**208/208**	**188/208**

Decker passes all 208 test cases. This is expected: Decker's deck construction is driven by a conservative static analysis over the program's call graph that approximates the set of functions reachable from any given call site. At runtime, the instrumentation inserted in compile-time activates each deck (via `mprotect`) before its functions are invoked and deactivates it after they return. Because the analysis is conservative, every function that could be needed at a given point is guaranteed to reside in an active deck. The result is a sound transformation where no code path that the program could exercise at runtime is removed.

Razor passes 205 of 208 test cases. The three failures are tar on a bz2-encoded input (the compression-specific code path was not exercised during training), and chown and rm on recursive directory traversals (the recursive traversal logic triggers trap instructions inserted by Razor's binary rewriter). All three failures reflect gaps in the training traces rather than corruption of the binary itself: the debloated program is structurally intact, but the heuristic path finder did not retain the specific code paths needed for these inputs.

Machete passes 188 of 208 test cases (90.4%). The 20 failures concentrate in grep (13 failures), chown (3), gzip (2), date (1), and mkdir (1). Unlike Razor's failures, Machete does not modify the binary; these failures are sandbox violations in which a page permission fault during a phase does not correspond to a transition from said phase. The grep failures are the most numerous, and all 10 failures on one of the two test files (`test1`) share the same root cause: `test1` is a 100 KB file consisting of a single 64 KB line with no newline terminators, whereas the profiling input (`train1`) is a conventional 65 000-line text file. The absence of line terminators forces grep into bulk-read and long-line processing code paths that are never exercised during profiling on well-structured input, causing the sandbox to encounter page accesses for which no learned phase exists. The remaining 3 grep failures occur on a second test file (`test2`) that is structurally similar to the training input but differs enough in content for three flag combinations (`-E`, `-n`, and bare pattern) to trigger untrained paths. Similarly, the chown failures arise because the test cases exercise recursive ownership changes that were not part of the profiling traces.

The correctness results illustrate a fundamental trade-off. Decker's reliance on static analysis yields a sound transformation at the cost of precision: it cannot remove code that the call graph

conservatively deems reachable. Razor and Machete both depend on the completeness of their execution traces, but Machete's enforcement is stricter. Where Razor silently retains untrained code paths through its heuristic path finder, Machete's phase-based restriction means that any deviation from the profiled behaviour manifests as a hard failure rather than silent over-approximation. Stronger enforcement requires broader trace coverage, and the 20 failures observed here motivate the profiling improvements discussed in Section 4.7.

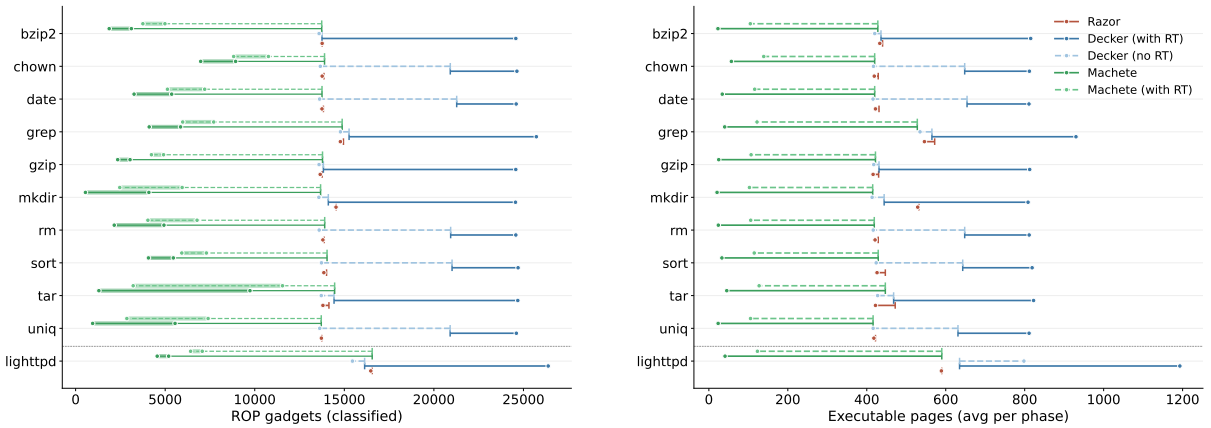
4.3 Security Posture Improvements

This section evaluates whether Machete materially reduces the attack surface available to an adversary, and how the reduction compares to that achieved by Razor and Decker. We examine five aspects: phase granularity of the learned sandboxes; the resulting reduction in executable pages and ROP gadgets; the effect of runtime support libraries; execute-only memory enforcement; per-class gadget structure and expressivity. We also go over the security metrics for the two auxiliary Machete-only targets.

Machete's learned FSMs produce between 2 and 4 phases for most targets. Despite this modest phase count, per-phase page availability varies substantially. Programs with distinct lifecycle stages show the widest variation: tar ranges from 17 to 67 executable pages across its 3 phases, chown from 51 to 63, and uniq from 10 to 37. Even programs with only 2 phases exhibit meaningful granularity accross phases (gzip: 22 to 28 pages; bzip2: 22 to 24). This variation is not manually designed; it emerges directly from the Blue-Fringe learner's state-merging decisions over the profiled execution traces.

Figure 4.1 presents the central result of this chapter. For each shared target, we compare the executable page count and classified ROP gadget count across the three tools and measure the impact each tool has on their respective baseline binary. For Machete and Decker, we additionally include the metrics for both with and without their runtime libraries. Each lollipop extends from the tool's own baseline (base of the horizontal line) to the debloated value (dot), so the length of the stick directly represents the magnitude of reduction.

Across all 10 shared utilities Razor, and Decker without runtime cluster at similar levels for both metrics. Razor's debloated binaries achieve a modest reduction in executable pages (1–11% depending on the target). The original `.text` section is retained in the binary as a read-only segment, and the debloated code is placed in a new executable section; since only the new section is mapped as executable, the reduction reflects the difference between the original code size and the debloated code. However, because Razor operates only on the application binary and leaves all shared libraries intact, the overall reduction is small relative to the total executable footprint. Decker without runtime achieves a similarly modest reduction on some targets (notably chown, date, rm, sort, and uniq) but leaves others essentially unchanged. Although Decker uses a custom linker script to arrange functions by deck into page-aligned groups, and activates only the decks needed at each point via `mprotect`, the conservatism of its static call-graph analysis limits the reduction: decks often contain more functions than a given execution actually requires, and all pages belonging to an active deck must be mapped as executable. Moreover, Decker operates only on application code and does not restrict library pages. By contrast, Machete learns which pages are actually accessed during profiled executions and restricts all file-backed pages (both program and library) on a per-phase basis. As a result, Machete reduces executable pages by 86–95% relative to its own baseline across all targets. Even in the worst-case (maximum) phase, Machete's page count is 2 to 5 times lower



(a) Classified ROP gadgets

(b) Executable pages

Figure 4.1: Attack surface reduction across shared targets. Lollipop length shows magnitude of reduction; dashed lines include runtime libraries. For machete, the difference between min and max phases are marked as a highlight on the machete line.

than either Razor or Decker. The ROP gadget reduction follows the same pattern: Machete achieves a 33–78% reduction (average 60%) even in the worst phase, while Razor and Decker achieve only single-digit percentage reductions.

A key contributor to Machete's advantage is that its enforcement covers all file-backed pages mapped into the process, including shared libraries. In most of the evaluated programs, the majority of executable pages belong to shared libraries (libc, libm, libpthread, and others). Razor rewrites the application binary but leaves libraries fully mapped and executable. Decker instruments application function calls but does not extend its deck partitioning to library code; library pages remain readable and executable throughout execution. Machete treats library pages identically to application pages: if a phase does not need a library page, that page is not accessible. This accounts for much of the observed reduction.

Both Decker and Machete rely on runtime support libraries that are mapped into the application's address space during enforcement, inflating the executable footprint beyond what the debloating strategy alone would produce. Figure 4.1 shows both the solid (what actually runs) and dashed (accounting view) variants for each tool, making the runtime contribution visible. For Decker, the runtime cost is substantial. `libdebloat_rt.so` contributes approximately 244 executable pages, and the additionally loaded `libm.so.6` adds another 151 pages. Including the runtime, the total rises to approximately 825 executable pages per phase – roughly 1.5 times the baseline average of 557. The runtime overhead thus dominates and largely cancels the modest application-level reduction that the deck mechanism provides without it.

For Machete, `libmachete_interp.so` adds 82 executable pages. Without the runtime, Machete's enforced phases contain 20 to 57 pages; with it, 103 to 139 pages. The runtime roughly doubles the page count, but even in the *with runtime* view, Machete remains 4 to 8 times lower than Decker with runtime. This asymmetry is by design: the Machete interpreter was written to be as small as possible and to be self-contained with no external dependencies, specifically to avoid the paradox of introducing additional mapped (which itself could be considered bloat) code while debloating.

Table 4.4: Average readable-executable (R+X) pages across shared targets.

Variant	Avg exec pages	Avg R+X pages
Razor debloated	445.2	445.2
Decker WPD (without runtime)	430.3	430.3
Machete enforced (with runtime)	114.6	0.0

Beyond quantitative reduction, Machete provides a qualitative security property that neither Razor nor Decker offers: execute-only memory (XoM) on application code. Under Razor and Decker, all executable pages are simultaneously readable (R+X). Recall that the attacker model adopted in Section 3.1 grants arbitrary read and write primitives on an already-vulnerable application. The write primitive enables code-reuse attacks such as ROP; the read primitive enables the attacker to scan code pages at runtime to discover gadgets dynamically, bypassing ASLR via just-in-time ROP. Under Razor and Decker, the R+X mapping of all executable pages leaves both primitives fully exploitable on code. Under Machete, application code pages are mapped with execute-only permissions, which neutralises the read primitive on code pages: the attacker can execute code that is mapped but cannot read it to locate gadgets. As Table 4.4 shows, the enforced variants have zero readable-executable application pages. This means Machete does not merely reduce the volume of executable code; it prevents the attacker from reading whatever executable code remains.

Total gadget counts, while informative, do not capture which attacker capabilities survive after debloating. Following the Gadget Set Analysis (GSA) methodology of Brown et al. [15], we break down the classified ROP gadgets into the 11 practical classes (load_const, move_reg, load_mem, store_mem, add, sub, and, or, xor, cond_branch, stack_pivot) and examine whether the reduction is uniform or concentrated in a few classes.

We use tar as the illustrative case because it has 3 phases with strong page variation aside from all tool variants being available for comparison. Figure 4.2 shows the per-class counts for Razor (baseline and debloated), Decker (baseline and WPD without runtime), and Machete (baseline and enforced). The top five rows (both baselines and debloated variants) are nearly identical across every class: neither Razor's binary rewriting nor Decker's deck-based restriction substantially changes the per-class gadget distribution for this target. Machete enforced (bottom row) is uniformly lower across every class, reducing each to roughly 30 to 60% of its own baseline value. Classes that are critical for exploit construction, such as add, move_reg, store_mem, and stack_pivot, are all reduced proportionally, and cond_branch drops from 7–8 gadgets to 3.

It is worth noting that while Machete reduces the number of gadgets in every class, it does not eliminate entire classes in most cases. Table 4.5 reports the expressivity metrics averaged across the 10 shared utility targets. Razor (baseline and debloated), Decker (baseline and WPD without runtime), and the Machete baseline all retain the full complement of practical ROP classes (11/11), Turing-complete classes (17/17), and approximately 7 out of 10 special-purpose gadget types. Machete enforced averages 10.9 practical ROP classes and 16.8 Turing-complete classes; in its tightest phases, it drops to 10/11 and 14/17 respectively (the mkdir and uniq sandboxes lose one practical class in their smallest phase).

Expressivity is therefore not where Machete's primary security advantage lies. The remaining gadget set in Machete's enforced phases retains near-full *theoretical* expressivity: an attacker who can chain gadgets can, in principle, still construct a Turing-complete payload. However, Machete

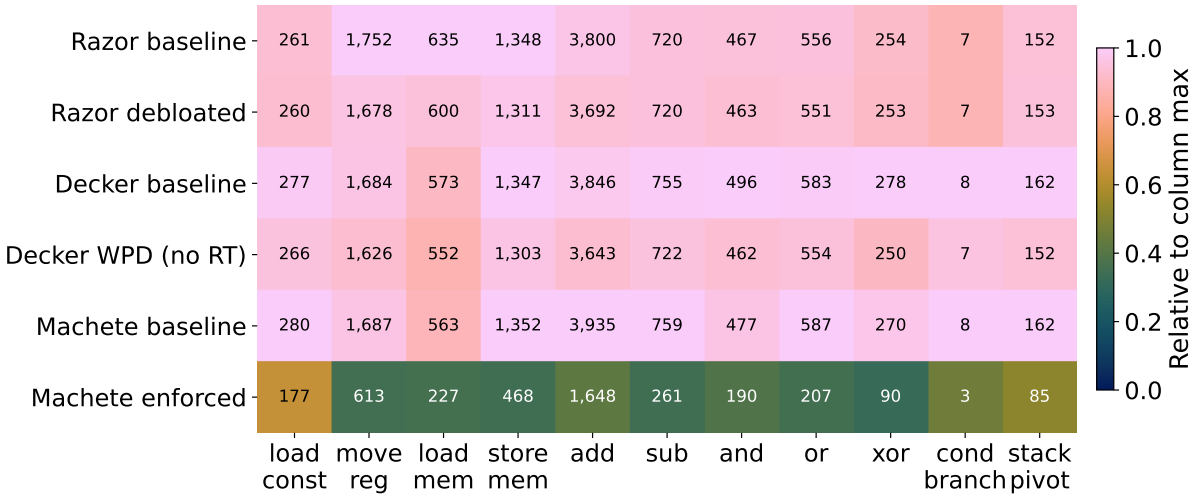


Figure 4.2: Per-class practical ROP gadget counts for tar. Colour normalized per column.

Table 4.5: Gadget expressivity averaged across shared targets. Machete shows cross-phase average and tightest-phase minimum.

Variant	Practical ROP (out of 11)	Turing-complete (out of 17)	SP gadget types (out of 10)
Razor baseline	11.0	17.0	7.4
Razor debloated	11.0	17.0	7.2
Decker baseline	11.0	17.0	7.8
Decker WPD (no RT)	11.0	17.0	7.1
Machete baseline	11.0	17.0	7.2
Machete enforced (min–max)	10.0–10.9	14.0–16.8	7.0

raises the practical difficulty of exploitation in two complementary ways. First, the quantitative narrowing of the gadget pool within each class reduces the number of alternative gadgets available, forcing the attacker to work with fewer and potentially less convenient building blocks. Second, execute-only enforcement prevents the attacker from scanning code pages to discover which gadgets are available in the current phase. Together, these properties mean that even though the theoretical expressivity of the remaining gadget set is largely preserved, the practical effort required to construct a working exploit is substantially higher.

We additionally evaluate two Machete-only targets, memcached and xz, that are not part of the shared comparison set. Both programs are multi-threaded, and their inclusion demonstrates a capability that the other tools do not support: Decker's architecture does not accommodate multi-threaded debloating because it relies on `mprotect` syscalls that affect the virtual memory permissions for all running threads, and Razor was not designed with thread-level enforcement in mind. In Machete, xz produces a 2-phase sandbox with strong separation: 52 pages during initialisation and compression setup, dropping to 21 pages during steady-state compression (a 60% reduction mid-execution). memcached produces a 4-phase sandbox with 27 to 43 active pages per phase, compared to 501 pages in the baseline. These results confirm that the phase-based narrowing

observed on the shared utility benchmarks generalises to multi-threaded compression pipelines and event-driven server software.

4.4 Runtime Overhead

The security improvements presented in the previous section are only practical if the enforcement overhead is acceptable. This section quantifies that cost across two classes of workload: CPU-bound compression utilities and syscall-dense server applications. In addition to the three debloating tools, we include K23 [53], the syscall interposition framework on which Machete is built. k23 is a low-overhead mechanism for intercepting system calls in user space using SUD and zpoline-style instruction rewriting. Machete uses k23 internally for two purposes: intercepting memory-management syscalls to maintain virtual-memory consistency across phase processes, and detecting syscall-triggered phase transitions. In the performance comparison, k23 runs the application under syscall interposition only – it intercepts every system call but performs no phase switching, page-permission enforcement, or debloating of any kind. Comparing k23 against Machete therefore isolates the overhead of Machete's process-level phase architecture from the baseline cost of syscall interposition alone.

Table 4.6 reports the results for all shared benchmarks. For compression (bzip2 and gzip on a 100 MB kernel tarball, 20 runs, 3 warmup iterations), all four tools are within a few percent of their own baselines. This is expected: compression workloads spend nearly all their time in tight computational loops with very few system calls or phase transitions, and Machete's enforcement mechanism is only invoked when a phase transition occurs. Programs that rarely transition therefore incur negligible overhead. Razor and Decker show slight speedups on some targets (Razor achieves –3.2% on gzip), likely due to reduced instruction-cache pressure from the smaller executable region. Machete's worst case is +3.8% on gzip, which is the only target where the overhead is measurably above noise; k23 is indistinguishable from baseline on both targets as they are not syscall-heavy workloads. The bottom line for CPU-bound workloads is that phase-based enforcement is essentially free, provided the learned sandbox properly separates the program's prologue and epilogue from its compute-heavy core.

lighttpd presents a completely different picture. Each HTTP request triggers a sequence of system calls (accept, read, write, sendfile, close) that may cross phase boundaries, making the enforcement overhead proportional to the rate of phase transitions rather than to the total runtime. Razor preserves throughput well (–1.6%) because the debloated binary is standalone and requires no runtime interposition. Decker incurs a modest –3.3% overhead; its compiler-inserted `mprotect` calls at deck boundaries are relatively infrequent compared to the per-request syscall rate. k23 incurs –7.4% overhead, representing the cost of syscall interposition without any phase switching. Machete's 2-phase sandbox reduces throughput by 66.6% (34,344 rps versus a 102,982 rps baseline), and the 3-phase sandbox by 90.3% (10,008 rps). The gap between k23′s 7.4% and Machete's 66.6% is attributable to Machete's multi-process phase-switching mechanism: segfault-based transition detection, cross-process context switching, and the VM consistency protocol that keeps all phase processes synchronised.

Several factors contribute to this overhead, and it is useful to distinguish them. First, *transition detection* relies on segmentation faults for some transitions: when the application touches a page that is not permitted in the current phase, the kernel delivers a SIGSEGV through its signal-handling path, which incurs the non-trivial overhead of OS signal delivery before Machete's interpreter

Table 4.6: Shared benchmark performance. Overhead is relative to each tool's own baseline.

Target	Metric	Variant	Baseline	Observed	Overhead
bzip2	runtime	Razor debloated	20.40 s	20.27 s	–0.6%
		Decker	9.20 s	9.18 s	–0.2%
		Machete	9.12 s	9.13 s	+0.1%
		k23	9.12 s	9.10 s	–0.2%
gzip	runtime	Razor debloated	6.13 s	5.93 s	–3.2%
		Decker	3.16 s	3.18 s	+0.7%
		Machete	3.21 s	3.34 s	+3.8%
		k23	3.21 s	3.21 s	–0.0%
lighttpd	throughput	Razor debloated	89,371 rps	87,967 rps	–1.6%
		Decker	88,980 rps	86,010 rps	–3.3%
		k23	102,982 rps	95,373 rps	–7.4%
		Machete 2-phase	102,982 rps	34,344 rps	–66.6%
		Machete 3-phase	102,982 rps	10,008 rps	–90.3%

regains control. Second, all *phase switching* requires a cross-process context transfer: the interpreter saves the complete register state to shared memory, wakes the target phase process via `futex`, and puts the current process to sleep. This process on its own does not incur too much overhead, the only slow operation on the chain is the `futex` wake syscall which incurs a context switching overhead. Each phase transition pays the phase switching cost unconditionally and the *transition detection* only for some transitions. For a server processing thousands of short-lived requests per second, these per-transition costs accumulate rapidly, especially if there is a segfault based transition on the hot-path.

The 2-phase versus 3-phase comparison is instructive. The 3-phase sandbox introduces an additional phase boundary per request lifecycle, which roughly triples the per-request enforcement cost. This demonstrates that the overhead scales with transition frequency, not with the number of phases. The comparison also reveals an explicit security/performance tradeoff at the heart of Machete's design: the same mechanism that provides strong per-phase isolation (process-level page-table separation with execute-only permissions) is also the source of its overhead on transition-heavy workloads. Reducing this cost without sacrificing XoM is an open problem and a primary target for future optimisation.

We additionally measure two Machete-only targets to broaden the performance picture. For xz (a 13-second compression run), overhead is indistinguishable from noise for both Machete (–0.1%) and k23 (–0.9%), confirming that CPU-bound workloads with few phase transitions incur negligible enforcement cost. For memcached (a key-value server under 30 seconds of sustained load), Machete incurs –2.7% overhead and k23 –1.7%. Unlike lighttpd, memcached absorbs enforcement with low overhead despite being a long-running server. The difference lies in the learned finite state machine: memcached's architecture provides a more clear structure for Machete to learn, where different threads have a specific subset of responsibilities that allow for easy segregation into distinct phases. This confirms that the predictor of Machete's overhead is the state machine that it runs, and not the number of system calls or program's utility. Table 4.7 reports the full auxiliary results.

Table 4.7: Auxiliary Machete-only performance results.

Target	Variant	Median	Mean	Std dev	Overhead
xz	Baseline	13.02 s	13.06 s	0.22 s	—
	Machete	13.00 s	13.03 s	0.14 s	–0.1%
	k23	12.90 s	12.96 s	0.19 s	–0.9%
memcached	Baseline	296,067 ops/s	298,690 ops/s	292,029 ops/s	—
	Machete	288,003 ops/s	296,782 ops/s	279,260 ops/s	–2.7%
	k23	291,086 ops/s	293,890 ops/s	286,464 ops/s	–1.7%

4.5 Sandbox Generation Parameter Sensitivity

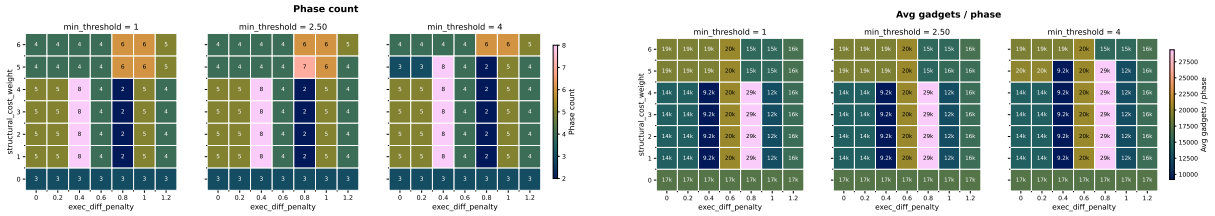
The previous sections evaluated Machete under fixed sandbox configurations. A natural follow-up question is whether the learned sandbox is fragile with respect to the scoring coefficients used during state merging, or whether the parameter space admits stable regions from which good sandboxes can reliably be obtained. More phases generally improve security by restricting each phase's executable footprint, but they also increase the number of runtime transitions and therefore the enforcement overhead. The scoring parameters give the operator a way to navigate this tradeoff.

To investigate, we swept two parameter families over `lighttpd`, the shared target with the richest feature set. For each sampled configuration, we derived a complete sandbox and measured its phase count, average disabled executable pages per phase, and average gadget count per phase.

The first sweep varies `exec_diff_penalty` and `structural_cost_weight`, with `min_threshold` as a slice parameter. `exec_diff_penalty` controls how strongly the learner resists merging phases whose executable-page signatures differ; `structural_cost_weight` rewards merges that simplify the state machine. The relationship between `exec_diff_penalty` and the resulting phase count is non-monotone. At very low values, the learner merges too aggressively, collapsing unrelated phases together, which paradoxically makes later convergence harder and produces a moderate number of broad phases. Around `exec_diff_penalty` ≈ 0.4, a peak emerges where the penalty is just strong enough to preserve meaningful code-page distinctions without over-constraining the search, yielding up to 8 phases. Above that point, increasingly aggressive penalties prevent merges, and the phase count drops again. `structural_cost_weight` has a more limited role and has a smaller influence on the resulting sandbox. From these results we pick the default values at `exec_diff_penalty = 0.5`, `structural_cost_weight = 5.0`, and `min_threshold = 3.0` as they result in smaller sandboxes. This makes it more likely to maintain correctness and reduces the overhead, but as we saw in the results from Section 4.3, still results in considerable improvements to the application's security.

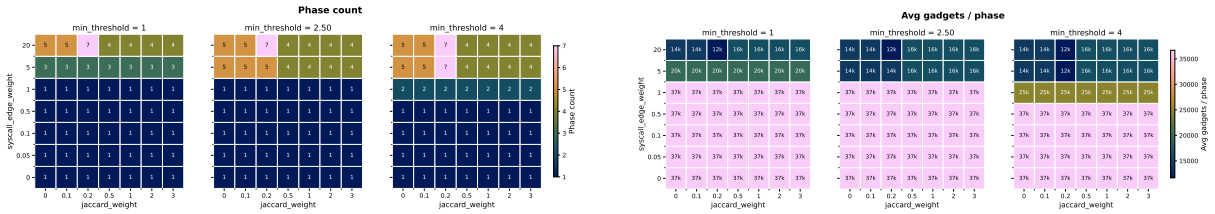
Figure 4.3 shows the non-monotone phase-count relationship alongside the resulting per-phase gadget exposure. The peak at `exec_diff_penalty` ≈ 0.4 is visible across all `structural_cost_weight` values above zero. Configurations with more phases consistently expose fewer gadgets in each individual phase, which is the metric that matters under the attacker model adopted in this thesis. The parameter space contains optima around `exec_diff_penalty` ≈ 0.4 and 0.8 from which the richest sandboxes are reliably obtained; the current defaults sit in between those optima to achieve a reasonable balance between performance and correctness with security.

The second parameter, `syscall_edge_weight`, serves a different purpose. Rather than controlling how many phases the learner produces, it controls *what kind of events* drive transitions between



(a) Learned phase count (b) Average gadgets per phase

Figure 4.3: Effect of `exec_diff_penalty` and `structural_cost_weight` on the learned `lighttpd` sandbox at three `min_threshold` slices.



(a) Learned phase count (b) Average gadgets per phase

Figure 4.4: Effect of `jaccard_weight` and `syscall_edge_weight` on the learned `lighttpd` sandbox at three `min_threshold` slices.

them. `syscall_edge_weight` penalises merges that would destroy transitions triggered by system calls. We compare `syscall_edge_weight` against `jaccard_weight` as it is an umbrella parameter that penalizes difference accross all page types. A high value preserves syscall-driven transitions in the learned state machine; a low value allows the learner to collapse them, often reducing the sandbox to a single phase. This distinction matters for performance: syscall-triggered transitions are detected through the interposition mechanism and do not require a segmentation fault, making them substantially cheaper than page-fault-triggered transitions. Steering the sandbox toward syscall-driven boundaries therefore improves enforcement performance without necessarily reducing the number of phases. Moreover, favoring syscall transitions is beneficial because they often mark significant events in a program's timeline, for example a server calling `poll` on the socket once its initialization is complete.

Figure 4.4.a confirms that below `syscall_edge_weight` ≈ 5.0 virtually all configurations collapse to a single phase regardless of `jaccard_weight`. Above that threshold, the results stabilise, and `jaccard_weight` has only limited independent effect. The current defaults sit well inside the stable multi-phase region. The practical conclusion is that `syscall_edge_weight` should be kept high enough to preserve syscall-driven transitions, both for the security benefit of maintaining multiple phases and for the performance benefit of cheaper transition detection.

Taken together, the two parameter families are largely orthogonal: one controls *how many* phases the learner produces, the other controls *what kind of events* trigger transitions between them. Within each family, the parameter space contains stable pockets rather than knife-edge optima, which means that moderate changes to the coefficients do not cause abrupt changes in

sandbox quality. The current defaults sit in a stable but conservative region; for `lighttpd`, lowering `exec_diff_penalty` toward the 0.2–0.6 range recovers meaningfully richer phase structure. More broadly, the sweep demonstrates that the scoring parameters give an operator meaningful control over the security/performance tradeoff: more phases and syscall-driven transitions improve security, while fewer phases and cheaper transition types reduce enforcement overhead.

4.6 Discussion

The results draw a clear picture: Machete achieves substantially stronger attack-surface reduction than either Razor or Decker, and the gap is not marginal. Three design decisions account for this. First, temporal debloating allows Machete to restrict which code is executable at any given moment rather than making a single static decision at build time. A program that needs compression routines during startup but not during steady-state serving has those routines removed from the executable footprint mid-execution, which neither Razor's binary rewriting nor Decker's deck activation can express. Second, Machete debloats the entire dependency chain, not just the application binary. Most of the executable footprint in the evaluated targets resides in shared libraries (libc, libm, libpthread), which Razor and Decker leave fully mapped and executable; D-Linker [26] addresses library debloating but does not touch application code. Machete treats library pages identically to application pages, which is why the page-count reductions in Figure 4.1 are so much larger. Third, because Machete's sandbox is learned from dynamic traces rather than derived from a static call-graph analysis, it is less conservative: code paths that are reachable in theory but never exercised in practice are not retained. Even a single-phase Machete sandbox outperforms both predecessors on every shared target, and multi-phase enforcement widens the gap further.

Machete does not, however, increase the theoretical security of the application as measured by the GSA expressivity metrics. Table 4.5 shows that the enforced variants retain nearly the full complement of practical ROP classes, Turing-complete classes, and special-purpose gadget types. An attacker who can chain gadgets can, in principle, still construct a Turing-complete payload from the surviving gadget set. What Machete does is raise the practical difficulty of exploitation in two complementary ways. The quantitative narrowing within each gadget class (30 to 60% fewer gadgets per class in the evaluated targets) forces the attacker to work with fewer and potentially less convenient building blocks, where a single missing link in the chain can prevent a payload from being assembled. More importantly, execute-only enforcement prevents the attacker from reading code pages to discover which gadgets are available in the current phase. As Table 4.4 confirms, all enforced variants have zero readable-executable application pages. Together, the reduced gadget pool and the inability to scan for surviving gadgets mean that even though theoretical expressivity is largely preserved, the practical effort required to construct a working exploit is substantially higher.

The performance results show that Machete's overhead depends almost entirely on the learned sandbox, not on the program itself. We see that the overhead ranges between 3% to 66% depending on the sandbox the program is executed under. This is not a fixed cost: the parameter sensitivity analysis in the previous section demonstrated that the operator can navigate the security/performance tradeoff by adjusting the scoring coefficients, and that the parameter space contains stable regions rather than knife-edge optima. The memcached result reinforces this point: despite being a long-running server, its well-structured thread architecture produces a sandbox with infrequent transitions and correspondingly low overhead.

Two limitations warrant explicit acknowledgement. The first is correctness. Out of the three tools, Machete had the lowest test-case pass rate (188/208). Every failure traces back to the same root cause: the sandbox is derived entirely from dynamic traces, and code paths not exercised during profiling are not represented in the learned FSM. When the application encounters such a path at runtime, the phase-based restriction manifests as a hard failure rather than silent over-approximation. This is a fundamental limitation of a purely dynamic analysis approach, and it motivates the static-analysis integration discussed in Section 4.7. The second limitation is that Machete can produce sandboxes with high enforcement overhead on transition-heavy workloads, and tuning the scoring parameters to find a better operating point currently requires manual experimentation. Although the parameter space is tractable, automating this search remains future work.

4.7 Future Work

Machete is composed of two largely independent components: the sandbox generation pipeline, which learns a phase-structured FSM from execution traces, and the enforcement runtime, which enforces the sandbox at execution time. Both can be improved independently, and the evaluation results point to specific directions for each.

The sandbox generation component is the most likely to benefit from near-term improvements, because it is the root cause of both the correctness failures and the overhead problems identified above. The most promising direction is to integrate static analysis into the learning pipeline. A static call-graph or control-flow analysis could identify code paths that are reachable but were not exercised during profiling, allowing the learner to assign them to appropriate phases rather than leaving them unrepresented. This would directly address the correctness gap: the 20 test-case failures we observed all involved code paths that a static analysis would have flagged as reachable. Beyond correctness, static analysis would also enable two performance improvements. First, transition points could be inserted at compile time as instrumented gates rather than detected at runtime via segmentation faults, eliminating the signal-delivery overhead that dominates the per-transition cost on lighttpd. Second, a static analysis that identifies hot paths and loop boundaries could inform the learner's placement of phase transitions, steering them away from high-frequency call sites and toward natural program lifecycle boundaries.

The enforcement runtime needs to be extended to support a wider range of application architectures. The current implementation was designed for modern event-driven programs that rely on threads and futexes for concurrency. However, established server software such as nginx and OpenSSH uses multi-process architectures with signals as the primary event-delivery mechanism, which is orthogonal to the concurrency model that the runtime assumes. Supporting these programs would require the runtime to take over the event-delivery mechanism: intercepting signals destined for the application, performing any necessary phase transitions, and then dispatching into the appropriate signal handlers. The core enforcement machinery (per-process page tables, execute-only permissions, and the VM consistency protocol) would remain unchanged; the extension is primarily in the interposition layer that bridges the application's concurrency model to Machete's phase-switching architecture.

Conclusion

5

This work began from a structural observation: the executable footprint of modern software is significantly larger than what is required at runtime. Across typical deployments, only a subset of mapped code is ever exercised, yet the entire dependency set remains available for execution. This excess is not benign. Each additional executable page increases the density of usable gadgets for code-reuse attacks, particularly in widely deployed libraries such as the GNU C Library. Existing mitigations address orthogonal aspects of this risk. Address space layout randomisation obscures locations but is undermined by disclosure primitives; execute-only memory removes read access but leaves the executable set unchanged; and W⊕X prevents injection but does not constrain reuse. What remains unaddressed is selectivity: none of these mechanisms governs which code should be executable at a given point during execution.

Prior work has approached this problem from multiple angles, but with incomplete coverage. Techniques operating at the source or compiler level inherently exclude closed-source and legacy binaries. Binary rewriting systems such as Razor reduce application-level code but do not extend to shared libraries, leaving a substantial portion of the attack surface intact. Static-analysis-driven approaches such as Decker provide structured partitions but are limited by conservative call-graph approximations and similarly omit library code. Other systems target only specific layers, such as dependency loaders or kernel-level isolation mechanisms, trading generality or deployability for stronger guarantees. Across there is no solution that covers three main requirements that make a debloating framework effective: temporal enforcement, end-to-end coverage across both application and library code and easy deployment without kernel tweaks.

Machete was developed to address these limitations by treating executable code as a temporally scoped resource. Rather than attempting to statically eliminate all unused code, it derives phase-specific execution policies from observed behaviour and enforces them dynamically without requiring source access or kernel support. The system combines three components: a profiling stage that records page-level access patterns across executions; a learning stage that constructs a compact, phase-structured model of program behaviour; and a runtime that enforces these phases through process-level isolation with per-phase execute-only permissions over shared memory. This design enables fine-grained control over the executable set throughout execution while remaining deployable in standard user-space environments. The resulting system provides full dependency-chain debloating and demonstrates measurable reductions in exposed code and gadget availability, as validated against Razor and Decker across a diverse benchmark set.

The research questions posed in the introduction can now be answered directly:

- **Main RQ: To what extent can temporal memory-access policies reduce the executable attack surface of unmodified binaries without kernel modifications?** Substantially. Machete demonstrates that user-space enforcement via per-process page tables can achieve page-level temporal debloating with execute-only memory, a combination previously available only through kernel extensions. The approach is practical for workloads with infrequent phase transitions and shows a clear path to improvement for transition-heavy workloads through static-analysis-guided gate insertion.
- **RQ1: To what extent can execution traces be transformed into compact policies that generalise across different executions of the same program?** The Blue-Fringe / EDSM

learner produces compact FSMs from raw page-access traces. The parameter sensitivity analysis showed that the scoring-coefficient space contains stable pockets rather than strips of local optima, meaning the learned models are robust to moderate parameter changes. The main limitation is that purely dynamic traces do not cover unseen code paths, which led to 20 out of 208 correctness failures across the shared benchmark set. Static analysis integration is the clearest path to closing this gap.

- **RQ2: To what extent does combining debloating with execute-only memory reduce exposed executable pages and gadget availability?** Machete reduces executable pages by 86 to 95% relative to its own baseline across all shared targets. Even in the worst-case phase, Machete's page count is 2 to 5 times lower than either Razor or Decker. ROP gadgets drop by 33 to 78% per phase, with an average reduction of roughly 60%. Execute-only enforcement eliminates all readable-executable application pages (zero R+X pages across all targets), preventing runtime gadget discovery. The GSA expressivity metrics are largely preserved (10 to 11 out of 11 practical ROP classes), so theoretical attack capability is not eliminated, but the combination of fewer gadgets per class and unreadable code pages raises the practical bar for exploitation substantially. Full dependency-chain debloating, covering shared libraries alongside application code, is also a major advantage over tools that operate only on the binary itself.
- **RQ3: What runtime overhead does user-space enforcement of temporal memory policies incur?** For all types of workloads, overhead depends on the learned sandbox: lighttpd with a 2-phase sandbox incurs 66.6% throughput loss, while memcached with a 4-phase sandbox incurs only 2.7%. The predictor is transition frequency, not program type. The process-based phase architecture is not the source of overhead on transition-heavy workloads, it is the transition detection itself.

Concretely, this thesis contributes: (1) a segfault-based profiling technique that captures page-granularity memory access patterns for both single- and multi-threaded programs; (2) a modified Blue-Fringe / EDSM learner that infers phase-structured finite-state machines from these traces, with tunable scoring coefficients controlling phase granularity; (3) a user-space enforcement runtime that uses per-process page tables over shared memory to achieve per-phase execute-only permissions without kernel modifications; and (4) a comprehensive evaluation comparing Machete against Razor and Decker across correctness, security, overhead, and parameter sensitivity on 11 shared and 2 auxiliary targets.

Machete shows that the combination of temporal debloating and execute-only memory is achievable in user space and yields security improvements that static and single-phase approaches cannot match. The main open challenges are correctness coverage, where purely dynamic traces leave unseen code paths unrepresented, and enforcement overhead on transition-heavy workloads. Both are addressable through the static-analysis integration discussed in the preceding chapter. The pipeline is deliberately modular: the profiler, learner, and enforcement runtime communicate through standalone artefacts and can each be improved or replaced independently. This makes Machete not only a working system but a foundation for future work on user-space memory hardening.

References

[1] A. Quach, A. Prakash, and L. Yan, *Debloating Software Through Piece-Wise Compilation and Loading*, in *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 869–886.

[2] *CVE-2014-0160: Heartbleed*, https://www.cve.org/CVERecord?id=CVE-2014-0160.

[3] *CVE-2021-44228: Log4shell*, https://www.cve.org/CVERecord?id=CVE-2021-44228.

[4] *CVE-2016-3714: Imagemagick Shell Injection*, https://www.cve.org/CVERecord?id=CVE-2016-3714.

[5] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, *Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization*, in *2013 IEEE Symposium on Security and Privacy* (2013), pp. 574–588.

[6] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, *RAZOR: A Framework for Post-deployment Software Debloating*, in *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 1733–1750.

[7] C. Porter, S. Khan, and S. Pande, *Decker: Attack Surface Reduction via On-Demand Code Mapping*, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ACM, Vancouver BC Canada, 2023), pp. 192–206.

[8] F. Rommel, C. Dietrich, A. Ziegler, I. Ostapyshyn, and D. Lohmann, *Thread-Level Attack-Surface Reduction*, in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, And Tools for Embedded Systems* (ACM, Orlando FL USA, 2023), pp. 64–75.

[9] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, *Enforcing Least Privilege Memory Views for Multithreaded Applications*, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Vienna Austria, 2016), pp. 393–405.

[10] J. Landsborough, S. Harding, and S. Fugate, *Removing the Kitchen Sink from Software*, in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (ACM, Madrid Spain, 2015), pp. 833–838.

[11] H. Shacham, *The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (On the X86)*, in *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007), pp. 552–561.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, *Jump-Oriented Programming: A New Class of Code-Reuse Attack*, in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), pp. 30–40.

[13] M. Alhanahnah, Y. Boshmaf, and A. Gehani, SoK: Software Debloating Landscape and Future Directions, (2024).

[14] M. Ali et al., *SoK: A Tale of~Reduction, Security, And~Correctness – Evaluating Program Debloating Paradigms and~Their Compositions*, in *Computer Security – ESORICS 2023*, edited by G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis (Springer Nature Switzerland, Cham, 2024), pp. 229–249.

[15] M. D. Brown, A. Meily, B. Fairservice, A. Sood, J. Dorn, E. Kilmer, and R. Eytchison, A Broad Comparative Evaluation of Software Debloating Tools, (2024).

[16] J. A. Navas and A. Gehani, OCCAM-v2: Combining Static and Dynamic Analysis for Effective and Efficient Whole-Program Specialization, Commun. ACM **66**, 40 (2023).

[17] M. Ali, R. Habib, A. Gehani, S. Rahaman, and Z. Uzmi, *BLADE: Scalable Source Code Debloating Framework*, in *2023 IEEE Secure Development Conference (Secdev)* (2023), pp. 75–87.

[18] H. Koo, S. Ghavamnia, and M. Polychronakis, *Configuration-Driven Software Debloating*, in *Proceedings of the 12th European Workshop on Systems Security* (ACM, Dresden Germany, 2019), pp. 1–6.

[19] C. Porter, S. Khan, K. Ni, and S. Pande, Combined Static Analysis and Machine Learning Prediction for Application Debloating, (2024).

[20] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, *Effective Program Debloating via Reinforcement Learning*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Toronto Canada, 2018), pp. 380–394.

[21] I. Wodiany, A. Pop, and M. Luján, LeanBin: Harnessing Lifting and Recompilation to Debloat Binaries, (2024).

[22] Y. Chen, T. Lan, and G. Venkataramani, *DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries*, in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (ACM, Dallas Texas USA, 2017), pp. 23–29.

[23] M. Mansouri, J. Xu, and G. Portokalidis, *Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs*, in *Proceedings of the ACM Asia Conference on Computer and Communications Security* (ACM, Melbourne VIC Australia, 2023), pp. 259–273.

[24] Z. Huang, *Debloating Feature-Rich Closed-Source Windows Software*, in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE, Rovaniemi, Finland, 2024), pp. 400–405.

[25] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, *Nibbler: Debloating Binary Shared Libraries*, in *Proceedings of the 35th Annual Computer Security Applications Conference* (ACM, San Juan Puerto Rico USA, 2019), pp. 70–83.

[26] J. He, P. Hou, J. Yu, J. Qi, Y. Sun, L. Li, R. Zhao, and Y. Wu, D-Linker: Debloating Shared Libraries by Relinking From Object Files, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **43**, 3768 (2024).

[27] A. Ziegler, J. Geus, B. Heinloth, T. Hönig, and D. Lohmann, Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries, ACM Transactions on Embedded Computing Systems **18**, 1 (2019).

[28] I. Agadakos, N. DeMarinis, B. Shteinfeld, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, Large-Scale Debloating of Binary Shared Libraries, Digital Threats: Research and Practice **1**, 19:1 (2020).

[29] H. Zhang, M. Ren, Y. Lei, and J. Ming, *One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries*, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, Lausanne Switzerland, 2022), pp. 255–270.

[30] Z. Hu, S. Lee, and M. Peinado, *Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis*, in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Copenhagen Denmark, 2023), pp. 1994–2008.

[31] H. Zhang, M. Alhanahnah, P. Leitner, and A. Ali-Eldin, The Cure Is in the Cause: A Filesystem for Container Debloating, (2025).

[32] H. Zhang and A. Ali-Eldin, The Hidden Bloat in Machine Learning Systems, (2025).

[33] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, *You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code*, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (ACM, Scottsdale Arizona USA, 2014), pp. 1342–1353.

[34] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, *NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64*, in *2017 IEEE Symposium on Security and Privacy (SP)* (IEEE, San Jose, CA, USA, 2017), pp. 304–319.

[35] J. Gionta, W. Enck, and P. Ning, *HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities*, in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (ACM, San Antonio Texas USA, 2015), pp. 325–336.

[36] T. Hornetz, L. Gerlach, and M. Schwarz, Lixom: Protecting Encryption Keys with Execute-Only Memory, (n.d.).

[37] C. Luo, J. Ming, M. Xie, G. Peng, and J. Fu, *Retrofitting XoM for Stripped Binaries Without Embedded Data Relocation*, in *Proceedings 2025 Network and Distributed System Security Symposium* (Internet Society, San Diego, CA, USA, 2025).

[38] M. Schink and J. Obermaier, *Taking a Look into Execute-Only Memory*, in *13th USENIX Workshop on Offensive Technologies (WOOT 19)* (2019).

[39] T. Hornetz and M. Schwarz, PortPrint: Identifying Inaccessible Code with Port Contention, Proceedings of the Microarchitecture Security Conference (2025).

[40] K. Bhat, E. Van Der Kouwe, H. Bos, and C. Giuffrida, *ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, Providence RI USA, 2019), pp. 545–558.

[41] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, *ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)*, in *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 1221–1238.

[42] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, *You Shall Not (By)pass!: Practical, Secure, And Fast PKU-based Sandboxing*, in *Proceedings of the Seventeenth European Conference on Computer Systems* (ACM, Rennes France, 2022), pp. 266–282.

[43] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, *Jenny: Securing Syscalls for PKU-based Memory Isolation Systems*, in *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 936–952.

[44] W. Blair, W. Robertson, and M. Egele, *ThreadLock: Native Principal Isolation Through Memory Protection Keys*, in *Proceedings of the ACM Asia Conference on Computer and Communications Security* (ACM, Melbourne VIC Australia, 2023), pp. 966–979.

[45] M. Unterguggenberger, L. Lamster, D. Schrammel, M. Schwarzl, and S. Mangard, *TME-Box: Scalable In-Process Isolation Through Intel TME-MK Memory Encryption*, in *Proceedings 2025 Network and Distributed System Security Symposium* (Internet Society, San Diego, CA, USA, 2025).

[46] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, *Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust*, in *2023 IEEE Secure Development Conference (SecDev)* (2023), pp. 54–66.

[47] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, *Secure and Efficient in-Process Monitor (And Library) Protection with Intel MPK*, in *Proceedings of the 13th European Workshop on Systems Security* (ACM, Heraklion Greece, 2020), pp. 7–12.

[48] A. Bittau, P. Marchenko, M. Handley, and B. Karp, *Wedge: Splitting Applications into Reduced-Privilege Compartments*, in *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (2008), pp. 309–322.

[49] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij, *muSwitch: Fast Kernel Context Isolation with Implicit Context Switches*, in *2023 IEEE Symposium on Security and Privacy (SP)* (IEEE, San Francisco, CA, USA, 2023), pp. 2956–2973.

[50] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, GHUMVEE: Efficient, Effective, and Flexible Replication, Foundations and Practice of Security **7743**, 261 (2013).

[51] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, *Zpoline: A System Call Hook Mechanism Based on Binary Rewriting*, in *2023 USENIX Annual Technical Conference (ATC 23)* (2023), pp. 293–300.

[52] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, *System Call Interposition Without Compromise*, in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (IEEE, Brisbane, Australia, 2024), pp. 183–194.

[53] J. M. Gómez Moreno, V. Moutafis, A. Dionysiou, F. Kuipers, G. Smaragdakis, B. Coppens, and A. Voulimeneas, *Clair Obscur: The Light and Shadow of System Call Interposition – From Pitfalls to Solutions with K23*, in *Proceedings of the 26th International Middleware Conference* (ACM, Vanderbilt University Nashville TN USA, 2025), pp. 241–255.

[54] W. Blair, F. Araujo, T. Taylor, and J. Jang, *Automated Synthesis of Effect Graph Policies for Microservice-Aware Stateful System Call Specialization*, in *IEEE Symposium on Security and Privacy* (2024).

[55] A. Rosti, S. Volckaert, M. Franz, and A. Voulimeneas, *I'll Be There for You! Perpetual Availability in the A8 MVX System*, in *Annual Computer Security Applications Conference (ACSAC)* (2024), pp. 520–533.

[56] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, *A Sense of Self for Unix Processes*, in *Proceedings 1996 IEEE Symposium on Security and Privacy* (IEEE Comput. Soc. Press, Oakland, CA, USA, 1996), pp. 120–128.

[57] N. Walkinshaw, R. Taylor, and J. Derrick, Inferring Extended Finite State Machine Models from Software Executions, Empirical Software Engineering **21**, 811 (2016).

[58] D. Angluin, Inductive Inference of Formal Languages from Positive Data, Information and Control **45**, 117 (1980).

[59] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars* (Cambridge University Press, Cambridge, UK, 2010).

[60] J. Oncina and P. García, Identifying Regular Languages in Polynomial Time, Advances in Structural and Syntactic Pattern Recognition **5**, 99 (1992).

[61] K. J. Lang, B. A. Pearlmutter, and R. A. Price, *Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm*, in *Grammatical Inference (ICGI)*, Vol. 1433 (Springer, 1998), pp. 1–12.

[62] M. Abadi, M. Budiu, '. Erlingsson, and J. Ligatti, Control-Flow Integrity: Principles, Implementations, and Applications, ACM Transactions on Information and System Security **13**, 1 (2009).

[63] M. Zhang and R. Sekar, *Control Flow Integrity for {} Binaries*, in *22nd USENIX Security Symposium (USENIX Security 13)* (2013), pp. 337–352.

[64] M. Payer, A. Barresi, and T. R. Gross, Fine-Grained Control-Flow Integrity Through Binary Hardening, Detection of Intrusions and Malware, And Vulnerability Assessment **9148**, 144 (2015).

[65] S. Ilahi, A. Omotosho, and C. Hammer, *ShadowGuard: Cryptographic Shadow Stack Protection with XOR Obfuscation and HMAC Integrity*, in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)* (IEEE, São Paulo, Brazil, 2025), pp. 119–129.

[66] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide, (2024).

[67] A. Soubki and J. Heinz, Benchmarking State-Merging Algorithms for Learning Regular Languages, (n.d.).

[68] J. Lettner, D. Song, T. Park, S. Volckaert, P. Larsen, and M. Franz, PartiSan: Fast and Flexible Sanitization via Run-time Partitioning, (2018).

[69] M. D. Brown and S. Pande, *Is Less Really More? Towards Better Metrics for~Measuring Security Improvements Realized Through Software Debloating*, in *Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test* (USENIX Association, Santa Clara, CA, USA, 2019).

Scoring Feature Definitions

A

Table A.1: Pre-score feature definitions.

Feature	Definition	Intuition
$J(r,b)$	$\lvert\mu(r) \cap \mu(b)\rvert - \lvert\mu(r) \cup \mu(b)\rvert$	Jaccard similarity of the enabled-item signatures. Higher values indicate that the two classes already share most of their items and are likely semantically equivalent.
$D_{\text{type}}(r,b)$	$\lvert\{i \in \mu(r) \triangle \mu(b) : \text{Op}(i) = \text{Execute}\}\rvert$	Number of *execute* items in the symmetric difference. Penalises merges between classes that differ in which code pages they enable, since these distinctions directly affect the attack surface.
$G_{\text{struct}}(r,b)$	$\lvert F(r)\rvert + \lvert F(b)\rvert - \lvert F(r) \cup F(b)\rvert$	Overlap between outgoing frontier edges. Rewards merges that collapse redundant structural transitions, reducing the total number of edges in the learned FSM.
$G_{\text{sys}}(r,b)$	$\lvert\{(c,D) \in F(r) \cup F(b) : \exists i \in D, \text{Op}(i) = \text{Syscall}\}\rvert$	Number of frontier edges carrying at least one syscall trigger item. Penalises the loss of syscall-bearing transitions, which are cheap to enforce at runtime and useful as phase discriminators.

This appendix gives the precise definitions of the scoring features used during Blue-Fringe merge selection (see Line 3.1). Both the pre-score and the post-score are weighted linear combinations of the form

$$s(r,b) = \sum_i w_i \cdot f_i(r,b) \tag{A.1}$$

where r is a RED class, b is a BLUE class, and w_i are configurable coefficients. The features f_i are defined below. Throughout, $\mu(c)$ denotes the enabled-item signature of class c, and $\text{Op}(i)$ extracts the operation class of item i.

AA Pre-score features

The pre-score is evaluated *before* stabilisation and is used only to rank RED candidates for a given BLUE class.

The pre-score is then:

$$s_{\text{pre}}(r,b) = \alpha_0 + \alpha_1 \cdot J(r,b) - \alpha_2 \cdot D_{\text{type}}(r,b) + \alpha_3 \cdot G_{\text{struct}}(r,b) - \alpha_4 \cdot G_{\text{sys}}(r,b) \tag{A.2}$$

AB Post-score features

The post-score is evaluated *after* stabilisation (Line 3.20) and therefore captures cascade effects that are invisible to the pre-score.

The post-score is then:

Table A.2: Post-score feature definitions. All quantities are computed over the stabilised hypothesis.

Feature	Definition	Intuition
$J(r, b)$	As above	Re-evaluated on the merged class $c = r \cup b$ and its neighbourhood after stabilisation.
$N_{\text{cascade}}(r, b)$	Number of cascade merges triggered during stabilisation	Each cascade merge collapses a pair of classes that were previously distinct. High values indicate that the initial merge has far-reaching structural consequences.
$N_{\text{removed}}(r, b)$	Number of items removed from transition domains during stabilisation	Items removed from domains shrink the set of observable triggers. High values indicate that the merge obscures behavioural distinctions.
$D_{\text{cascade}}(r, b)$	$\sum_{(x,D,y)\in E_{\text{cascaded}}} \lvert D\rvert$	Total domain size of edges affected by cascade merges. Quantifies how much frontier structure was disrupted by induced merges.
$\Delta_{\text{struct}}(r, b)$	Net reduction in non-syscall outgoing edges after stabilisation	Positive values indicate that the merge simplified the structural transition graph.
$\Delta_{\text{sys}}(r, b)$	Net reduction in syscall-bearing outgoing edges after stabilisation	Penalised because losing syscall transitions reduces phase discrimination at negligible runtime cost.

$$s_{\text{post}}(r, b) = \beta_0 + \beta_1 \cdot J(r, b) - \beta_2 \cdot N_{\text{cascade}}(r, b) \tag{A.3}$$

$$- \beta_3 \cdot N_{\text{removed}}(r, b) - \beta_4 \cdot D_{\text{cascade}}(r, b) \tag{A.4}$$

$$+ \beta_5 \cdot \Delta_{\text{struct}}(r, b) - \beta_6 \cdot \Delta_{\text{sys}}(r, b) \tag{A.5}$$

AC Coefficient mapping and defaults

The appendix formulas use generic Greek-letter coefficients. Table A.3 lists the correspondence between these coefficients, the implementation parameter names (as they appear in the scoring profile JSON and the command-line interface), and their default values. Note that α_2 (D_{type}) is a simplification: the implementation maintains separate per-type penalties for read, write, and execute items, which are summed and normalised by the total symmetric difference size. Because execute-item differences dominate the security-relevant behaviour, the feature definition above reports only the execute component; the full typed penalty is $(w_{\text{read}} \cdot d_{\text{read}} + w_{\text{write}} \cdot d_{\text{write}} + w_{\text{exec}} \cdot d_{\text{exec}})/ \lvert\mu(r) \triangle \mu(b)\rvert$, where all three weights default to 0.5. Similarly, β_4 in the post-score uses the same per-type penalty mechanism applied to the cascade-accumulated symmetric difference rather than a separate coefficient.

The default coefficient values were chosen by manual calibration on representative profiling traces of the shared benchmark targets, balancing the number of learned phases against the per-phase permission window size. The parameter sensitivity analysis in Section 4.5 systematically varies the most influential coefficients (`exec_diff_penalty`, `structural_cost_weight`,

Table A.3: Coefficient mapping between appendix notation and implementation parameters.

Coefficient	Parameter name	Default	Role
	Pre-score		
α_0	`merge_bonus`	8.0	Base bonus for any merge attempt
α_1	`jaccard_weight`	0.0	Reward for signature overlap (Jaccard similarity)
α_2	`exec_diff_penalty`	0.5	Typed diff penalty (see note above)
α_3	`structural_cost_weight`	5.0	Reward for structural simplification
α_4	`syscall_edge_weight`	20.0	Penalty for losing syscall-bearing transitions
	Post-score		
β_0	`merge_bonus`	8.0	Same base bonus as pre-score
β_1	`jaccard_weight`	0.0	Same Jaccard weight as pre-score
β_2	`cascade_penalty`	0.0	Penalty per cascade merge during stabilisation
β_3	`removal_penalty`	0.0	Penalty per item removed from transition domains
β_4	(typed diff)	0.5	Cascade typed diff penalty (same per-type weights as α_2)
β_5	`structural_cost_weight`	5.0	Reward for net structural edge reduction
β_6	`syscall_edge_weight`	20.0	Penalty for net syscall edge elimination
	Acceptance		
θ	`min_threshold`	3.0	Minimum post-score to accept a merge

`jaccard_weight`, and `syscall_edge_weight`) and confirms that the parameter space contains stable regions rather than knife-edge optima.